Universal Plug and Play Machine Models

U. Glässer, Y. Gurevich and M. Veanes
{glaesser, gurevich, margus}@microsoft.com

June 15, 2001

Technical Report
MSR-TR-2001-59

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract

Recently, Microsoft took a lead in the development of a standard for peer-to-peer network connectivity of various intelligent appliances, wireless devices and PCs. It is called the *Universal Plug and Play Device Architecture* (UPnP). We construct a high-level *Abstract State Machine* (ASM) model for UPnP. The model is based on the ASM paradigm for distributed systems with real-time constraints and is executable in principle. For practical execution, we use AsmL, the Abstract state machine Language, developed at Microsoft Research and integrated with Visual Studio and COM. This gives us an AsmL model, a refined version of the ASM model. The third part of this project is a graphical user interface by means of which the runs of the AsmL model are controlled and inspected at various levels of detail as required for e.g. simulation and conformance testing.

1 Introduction ... 3
2 The UPnP Protocol ... 4
2.1 Basic Properties ... 4
2.2 Sample UPnP Device ... 5
3 Abstract State Machine Model ... 6
3.1 Distributed Real-Time ASM ... 6
3.1.1 Components and Interfaces ... 7
3.1.2 Abstract Data Structures ... 8
3.1.3 TCP/IP Network and Protocols ... 9
3.1.4 Basic Agent Types ... 9
3.1.5 Discrete Time and Timeout Events ... 10
3.2 Addresses and Messages ... 11
3.2.1 Addresses ... 11
3.2.2 Representation of Messages ... 12
3.2.3 Sending and Receiving Messages ... 12
3.3 Abstract Protocol Model ... 13
3.3.1 Initial States ... 13
3.3.2 Network Model ... 13
3.3.3 Device Model ... 15
4 Executable Protocol Model and GUI ... 17
4.1 Gaphical User Interface ... 17
4.2 Executable Protocol Model ... 19
4.3 Communication Model ... 20
4.3.1 IP Address Space ... 20
4.3.2 Representation of Messages ... 20
4.3.3 Sending and Receiving Messages ... 21
4.4 Network Model ... 22
4.4.1 RunNetwork ... 22
4.4.2 DHCP Server ... 22
4.5 Control Point Model ... 23
4.5.1 Search for Devices ... 23
4.5.2 Processing Ads ... 24
4.5.3 Controlling Devices ... 24
4.6 Device Model ... 25
4.6.1 Addressing ... 25
4.6.2 Discovery ... 27
4.6.3 Description ... 28
4.6.4 Presentation ... 28
4.6.5 Eventing ... 28
4.6.6 Control ... 28
4.6.7 SERVICE Interface ... 29
5 Conclusions ... 29
Acknowledgements ... 30
References ... 30
6 Appendix I: AsmL model of CD Player ... 31
7 Appendix II: IServer Interface ... 43

1 Introduction

The group on Foundations of Software Engineering at Microsoft Research has developed a powerful specification language based on the notion of Abstract State Machines (ASMs) [4]. The language is called *AsmL*, the *Abstract state machine Language* [9]. AsmL is executable. Furthermore, it is integrated with Microsoft software development environment including Visual Studio and COM, Component Object Model [13]. AsmL supports specification and rapid prototyping of object oriented and component oriented software.

The main strength of ASMs in general and AsmL in particular is the precise, rigorously defined semantics. ASMs have been used to specify architectures, protocols, modeling languages, programming languages, and so on [10]. In particular, the International Telecommunication Union (ITU) recently approved an ASM-based formal semantics definition of SDL, the Specification and Description Language of ITU, as an official ITU-T standard [11]. AsmL was developed in order to deploy the ASM technology for industrial software development, in particular at Microsoft; see [12] for an overview.

Recently, Microsoft took a lead in the development of a standard for peer-to-peer network connectivity of various intelligent appliances, wireless devices and PCs. The current version of the standard is *Universal Plug and Play* (UPnP) *Device Architecture* V 1.0 defined in [1]; see also the website [2] of the UPnP Forum. Based on [1], we present in this document a high-level, executable ASM model of the protocol underlying the UPnP architecture. The model is concurrent, interactive, and real-time dependent. Here is how UPnP is described in [1]:

> *Universal Plug and Play is a distributed, open networking architecture that leverages TCP/IP and the Web technologies to enable seamless proximity networking in addition to control and data transfer among networked devices in the home, office and public spaces.*

This paper is a part of a larger study of distributed systems. Starting from an informal specification, like the UPnP standard, we construct a hierarchy of executable mathematical models called Abstract State Machines or ASMs. In this case, we construct two ASMs, a higher-level ASM described in Section 3, and a lower-level AsmL in Section 4. We cover most of the UPnP definition; there are no conceptual difficulties in covering the remainder.

What are executable mathematical models good for? Unlike traditional engineering disciplines, like mechanical or electrical engineering, systems engineering heavily relies on informal documentation. Such informal documentation is necessary and, as in the case of the UPnP definition, may be informative and useful. Still, informal documentation is informal and thus may be and often is ambiguous, incomplete, and even inconsistent. Properly constructed, mathematical models are consistent, avoid unintended ambiguity and are complete in the appropriate sense. Certain properties of the design can be proved mathematically. Furthermore, in contrast with informal documentation, our mathematical models are executable and so they can be used to explore and test the design. You can validate your models and generate test suites for conformance testing of your implementation. Let us emphasize that our mathematical models build on the given informal description. We fix loose ends, resolve ambiguities and inconsistencies, separate concerns, and so on. Gradually the given informal description gives rise to an executable mathematical model or to a hierarchy of such models.

In this paper, we concentrate on interoperability aspects rather than details of individual components. Components operate concurrently and interact with each other by exchanging messages over the

communication network. They use actuators and sensors to interact with the *external world*, which is the environment into which the whole system is embedded. The ASM paradigm allows us to combine *synchronous* as well as *asynchronous* computations. The component models themselves are parallel compositions of synchronously operating ASMs. The system as a whole is a composition of asynchronously operating components, called *agents*.

The Document Structure. Section 2 gives a brief overview of the UPnP protocol and illustrates a sample UPnP device. In Section 3 we introduce the higher-level UPnP machine model in several steps. In 3.1 we explain its overall structure and characteristic features, the underlying notions of concurrency and time, and its main components and interfaces. In 3.2 we define the basic communication primitives and data structures. In 0 we define the behavior model. In Section 4 we introduce the lower-level protocol model in AsmL and illustrate the tool environment for simulating the AsmL model. We start by treating basic properties of addressing and communication (Section 4.3). Next, we define the individual component models, namely: the network model(Section 4.4), the control point model (Section 4.5) and the device model (Section 4.6). Conclusions are presented in Section 5. Appendix I contains a detailed AsmL model of the sample UPnP device that is briefly described in Section 0.

Remark. This document is available in electronic form that allows automatic code extraction for feeding the AsmL compiler [9]. The AsmL parts are marked by a special document style so that they can easily be identified within the document.

2 The UPnP Protocol

In the given application context, we attempt to accurately reflect the abstraction level of the informal description of the UPnP Device Architecture as defined in [1]. Nonetheless, one wants to abstract from those details that are irrelevant for the understanding of the principle protocol behavior. To figure out what is relevant and what can be neglected is often not trivial and sometimes impossible without consulting the application domain experts. In our case these experts are the UPnP developers at Microsoft.

2.1 Basic Properties

We briefly summarize here basic characteristics of the UPnP architecture. Technically, this is a layered protocol architecture built on top of TCP/IP networks by combining various standard protocols, e.g. such as DHCP, SSDP, SOAP and GENA. It supports dynamic configuration of any number of *devices* offering *services* requested by *control points*. To perform certain control tasks, a control point needs to know what devices are available (i.e. reachable over the network) and what services these devices advertise. For a concrete example of a UPnP device see Section 2.2.

Restrictions. In general, the following restrictions apply:

- Devices may come and go at any time with or without prior notice. Consequently, there is no guarantee that a requested service is available in a given state or will become available in a future state.
- An available service may not remain available until a certain control task using this service has been completed.
- Control points and devices interact through exchange of messages over a TCP/IP network, where specific network characteristics (like bandwidth, dimension, reliability) are left unspecified. As such, communication is considered to be neither predictable nor reliable, i.e. message transfer is subject to arbitrary and varying delays, and certain messages may even get lost.

Basic Steps. The UPnP protocol defines 6 basic steps or phases. Initially, these steps are invoked one after the other in the order given below, but may arbitrarily overlap afterwards.

- Step 0: *Addressing* is needed for obtaining an IP address when a new device is added to a network.
- Step 1: *Discovery* informs control points about the availability of devices and their service.
- Step 2: *Description* allows control points to retrieve detailed information about a device and its capabilities.
- Step 3: *Control* provides mechanisms for control points to access and control devices through well-defined interfaces.
- Step 4: *Eeventing* allows control points to receive information about changes in the state of a service at run time.
- Step 5: *Presentation* enables users to retrieve information about a device as needed by for controlling the device.

2.2 *Sample UPnP Device*

As an example we consider a *CD player* [2]. In our model this device has two services, called `ChangeDisc`, `PlayCD`, where Figure 1 illustrates only the first one. The `ChangeDisc` allows a control point to add or remove discs from the CD player and to choose a disc to be placed on the tray. The complete service has the following basic actions (not requiring any arguments).

- AddDisc
- NextDisc
- PrevDisc
- RandomDisc
- OpenDoor
- CloseDoor
- ToggleDoor
- HasTrayDisc
- IsDoorOpen

The `PlayCD` service has the following actions.

- Play
- Pause
- Stop
- SetPlayProgram ONCE_RANDOM (*or* REPEAT_RANDOM *or* REPEAT_IN_ORDER)
- SelectTrack *number*
- NextTrack
- PrevTrack

The full AsmL model of the CD Player with these two services is given in Appendix I.

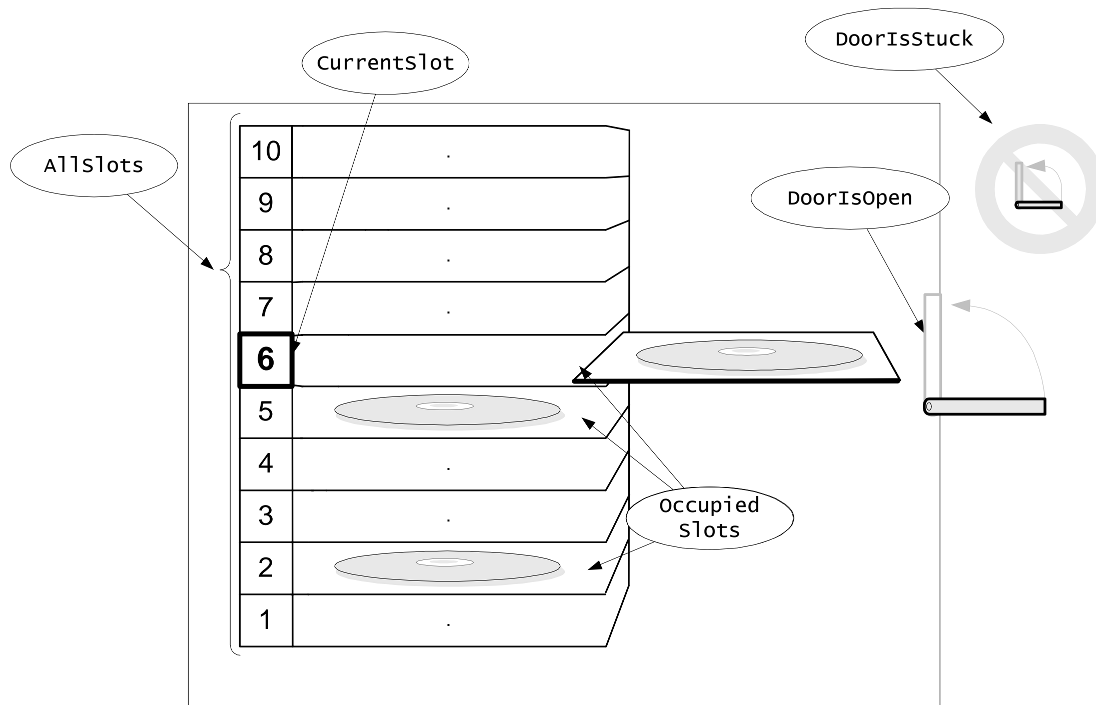

Figure 1: ChangeDisc service of a CD Player

3 Abstract State Machine Model

This section describes our ASM model of UPnP and the rational behind. The ASM model serves as a conceptual framework for dealing with system behavior at a more abstract level; nevertheless, it is closely related to the executable AsmL version, as described in Section 4, so that the translation into the latter becomes obvious. Conceptually, the ASM model is designed to meet two fundamental requirements on technical descriptions of complex software systems, namely:

- *robustness* provides the flexibility needed to extend and modify the model with a reasonable effort,
- *simplicity* avoids formalization overhead by stating behavior at the given level of abstraction, i.e. reflecting the view of the informal description.

Technically speaking, the model classifies as *distributed real-time ASM* and as such is based on fairly general notions of concurrency and time. Aiming at an intuitive understanding, we explain the underlying mathematical concepts in a rather informal style concentrating on those aspects that are relevant here. For a rigorous mathematical definition of the theory of Abstract State Machines, see the original literature [3,5].

3.1 Distributed Real-Time ASM

A reasonable choice for the construction of an abstract UPnP protocol model is a distributed real-time ASM consisting of an arbitrary number of concurrently operating and *asynchronously* communicating components. Intuitively, a component either represents a device, a control point or some fraction of the underlying communication network. With each component type we associate one or more interfaces such that any interaction between a component and any other component is restricted to actions and events as observable at these interfaces. Furthermore, actions and events in the external world as represented by the environment into which the system under consideration is embedded may affect the system behavior in various ways. For

instance, the transport of messages over the communication network is subject to delays and some messages may even get lost. Also, the system configuration itself may change as devices come and go. Such actions and events are basically unpredictable. We therefore introduce an additional GUI (cf. Section 4.1) that allows for user-controlled interaction with the external world. The overall organization of the model is illustrated in Figure 2.

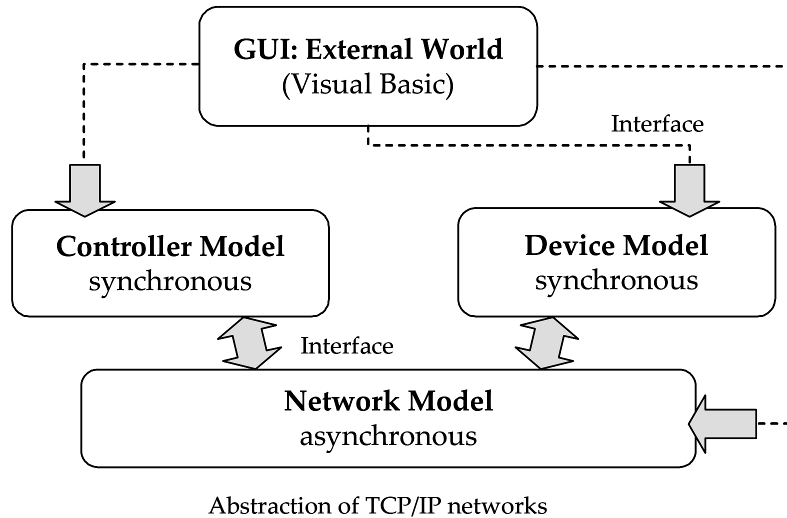

Figure 3: Overall organization of the distributed ASM model of UPnP.

At the component level, control points and devices are further decomposed, where each individual component splits into a collection of *synchronously* operating functional units. This decomposition is such that each of the resulting units participates in a different protocol step (cf. Section 2.1). Accordingly, we model control points and devices as parallel compositions of synchronously operating ASMs. For dealing with real-time constraints, we introduce a discrete notion of time for the abstract representation of global system time. In particular, we model *timeout events* through timer mechanisms (Section 0).

3.1.1 Components and Interfaces

We formulate dynamic properties of the UPnP protocol in terms of component interactions. Components operate autonomously and so that we can identify them with ASM *agents* in the distributed ASM. Agents come as elements of a dynamically growing and shrinking universe (or *domain*) `AGENT, where` we associate with each agent a program defining its behavior. A program consists of guarded update *rules* specifying state transitions through local updates on global states. We distinguish different types of agents according to different types of programs as represented by a static universe `PROGRAM`.

```
domain AGENT, static domain PROGRAM
```

In any given state of an ASM run, the behavior of an agent is well defined as stated by a unary dynamic function `program`. Being dynamic, this function allows introducing new agents at run time.

```
program : AGENT → PROGRAM, forall a ∈ AGENT: program(a) ∈ PROGRAM
```

Any interaction between the model and the external world, as observable at the respective interfaces, is reduced to interaction between two different categories of agents: (1) explicitly defined agents of the model, and (2)

implicitly given agents of the environment. The non-deterministic nature of environment agents naturally reflects the system view of the environment. However, this does not mean that environment behave in a completely unpredictable way; rather one can formulate reasonable *integrity constraints* on external actions and events where appropriate.

Roughly, one can characterize the model through the following basic properties:

- Initial State: An initial state specifies some finite collection of agents, which may grow and shrink dynamically over an ASM run.
- ASM Agents: There are three types of explicit agents, namely: control point agents, device agents and network agents.
- Concurrency: Agents operate concurrently. They interact by reading and writing shared locations of globally shared system states. The underlying semantic model regulates interaction between agents so that potential conflicts are resolved according to the definition of *partially ordered runs* [4].
- Instantaneous Reactions: Agents react instantaneously, i.e. they fire their rules as soon as they reach a state in which the rules are enabled. (Strictly speaking, one must assume here some non-zero delay to preserve the causal ordering of actions and events; though, this delay is immaterial from an application point of view.).
- Atomicity: Computation steps of agents are *atomic*, but, nevertheless, are considered as time-consuming actions.
- Discrete Time: System time is based on a discrete notion of time with time values being represented as real numbers.

3.1.2 Abstract Data Structures

In order to simplify modeling and to stay close to the informal understanding, we assume the presence of a rich background structure. In particular, we will be using *sets* and *maps* in our model. We may have sets of integers, maps from integers to strings, or even sets of such maps, etc. Both maps and sets may be viewed as aggregate entities and may be updated point-wise. For example, if `s` is a set of integers,

```
s : Set of Integer
```

say `s` is {1,2} in the current state then we may update `s` by firing the following parallel update in order to remove 2 from `s` and to add 3 to `s`.

```
s(2) := false
s(3) := true
```

In the state resulting from firing this rule, the value of `s` is {1,3}. If `m` is a map from integers to integers,

```
m : Map of Integer to Integer
```

that maps 2 to 3 and 4 to 5, then we may update `m` so that it maps 4 to 6 by firing the rule

```
m(4) := 6
```

We also have dynamic functions whose range consists of maps. For example, we may have a unary dynamic function `f` from integers to maps from integers to integers.

```
f : Integer → Map of Integer to Integer
```

If in the current state `f(1)` is a map from 2 to 3 and 4 to 5, in symbols `f(1)(2)=3` and `f(1)(4)=5`, then

we may update `f` (1) to map 4 to 6, just as we did with m above, by firing the rule

```
f(1)(4) := 6
```

Justification for the aggregate view of maps and sets in terms of standard ASMs is given in [15].

3.1.3 TCP/IP Network and Protocols

To model the network behavior, we define an abstraction of TCP/IP networks using standard network terminology [7]. Our network model is based on a distributed execution model reflecting the fact that a TCP/IP network usually consists of a (not further specified) collection of interconnected physical networks. However, we abstract here from topological details, e.g. how a global network is formed by interconnecting local networks using routers (or gateways); rather we describe the overall network behavior through a collection of concurrently operating *communicators*, each of which refers to some local network in conjunction with its adjacent routers. Conceptually, we separate behavior of the network and its routers (or gateways) from behavior of the hosts attached to this network as illustrated in Figure 2.

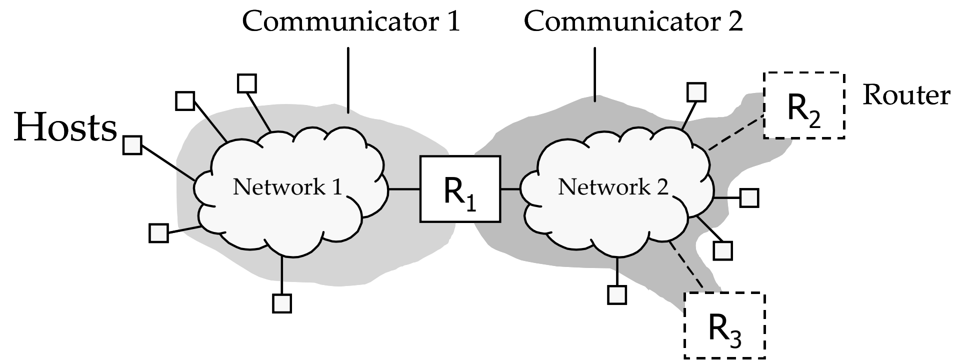

Figure 4: Communicators.

Based on the two standard transport level protocols, UDP (User Datagram Protocol) and TCP (Transmission Control Protocol), user level processes, or application programs, interact with each other by exchanging messages over the network. According to this view, there may be several application programs running on a single host. The *address* of an application program is given by the IP address of its host in conjunction with a unique protocol port number on this host. In our case, several control point programs may run on the same host. Devices, however, are considered as individual hardware units; therefore they are identified with the hosts on which they run. Collectively, we refer to control points and devices as *applications*.

3.1.4 Basic Agent Types

This section introduces various domains identifying the basic types of agents and gives an overview on how they are related with each other. We start with the main objects, namely: communicators, control points and devices.

```
domain COMMUNICATOR, domain CONTROL·POINT, domain DEVICE
```

DHCP Server Interface. The Dynamic Host Configuration Protocol (DHCP) enables automatic configuration of IP addresses when adding a new host to a network [8]. We model interaction between a DHCP server and the

DHCP client of a device explicitly only as far as the device side is concerned (cf. Section 3.3.3). The server side is abstractly represented through one or more *external* DHCP server agents whose behavior is left implicit. In our model, the DHCP server represents another type of application program.

```
domain DHCP·SERVER
```

We can now define AGENT as a derived domain, where we assume the three underlying domains COMMUNICATOR, CONTROL·POINT, DEVICE and DHCP·SERVER to be pairwise disjoint.

```
AGENT       ≡  APPLICATION ∪ COMMUNICATOR
APPLICATION ≡ CONTROL·POINT ∪ DEVICE ∪ DHCP·SERVER
```

An overview of the various agent types and the relations between them is presented in the form of an UML class diagram in Figure 3. In the subsequent sections, the keyword *me* will be used to identify the agent under consideration, i.e. as identified by a given context. In ASMs, classes are dynamic universes of objects. Consequently, a field f of type D of a class C can be viewed as a dynamic unary function from C to D, in symbols $f : C \rightarrow D$. For instance, a communicator has a field routingTable denoting a mapping from addresses to sets of communicators.

```
routingTable: COMMUNICATOR → Map of ADDRESS to (Set of COMMUNICATOR)
```

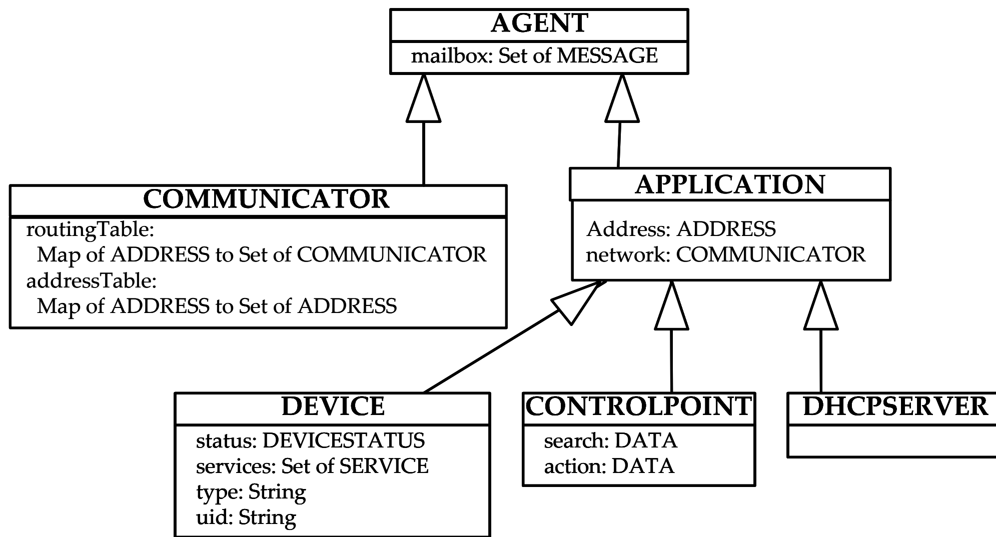

Figure 3: UML class diagram of the agents.

3.1.5 Discrete Time and Timeout Events

Time values are represented as real numbers by the elements of a linearly ordered domain TIME. We can assume that TIME ⊆ REAL and define the relation "≥" on time values through the corresponding relation on real numbers. A domain DURATION represents finite time intervals as differences between time values.

```
domain TIME, domain DURATION
```

Our notion of time is based on the view that we can only observe, but not control, how physical time evolves. Accordingly, we introduce a monitored, nullary function now taking values in TIME. Intuitively, now represents the global system time as measured by some discrete clock. One can reasonably assume that the values of *now* change monotonically over ASM runs.

```
monitored now : TIME initially startTime
```

An agent a may employ several distinct timers for different purposes, where each individual timer t has its own default duration effectively determining the expiration time when setting t. In a given state, a timer t is active if and only if its expiration time time(t) is greater than the value of now. Otherwise, t is called *expired*.

```
domain TIMER·TYPE ≡ {dhcpClient, discovery, …}
duration : AGENT → Map of TIMER·TYPE to DURATION
time : AGENT → Map of TIMER·TYPE to TIME
```

For a given timer t of agent a, the operation of setting t can thus be defined as follows.

```
SetTimer(a,t) ≡  time(a)(t):= now + duration(a)(t)
```

In a given state, a predicate Timeout indicates for given timer instance t and agent a whether the timer instance is active or has expired.

```
Timeout : AGENT → Map of TIMER·TYPE to BOOL, Timeout(a,t) = now ≥ time(a)(t)
```

3.2 Addresses and Messages

This section defines the representation of addresses and messages together with the mechanisms for sending and receiving messages. Our model abstractly reflects the view of the transport layer protocols TCP and UDP. At the given abstraction level, the only real difference between TCP and UDP is that the former is reliable whereas the latter provides a best-effort, connectionless packet delivery service, i.e. messages may get lost.

3.2.1 Addresses

We introduce a static universe ADDRESS of *IP addresses* that are extended by *protocol port numbers* to refer to the global TCP/UDP address space. For each application under consideration, a dynamic function address identifies an element from ADDRESS. A distinguished address called thisDevice is used as a source address for newly added devices that do not yet have an otherwise defined IP address.

```
static domain ADDRESS
address : APPLICATION → ADDRESS
```

When a new device is added to the network, it does not yet have an IP address, but uses its hardware address instead for communication with a DHCP server [8]. We abstractly represent hardware addresses as elements of a static domain HW·ADDRESS.

```
static domain HW·ADDRESS
hwAddress : DEVICE → HW·ADDRESS
```

3.2.2 Representation of Messages

Messages are uniformly represented as elements of a dynamic domain MESSAGE. Each message is of a certain *type* from the static domain MSG·TYPE. The message *type* determines whether a message is transmitted using UDP or TCP, though we do not make this distinction explicit.

```
domain MESSAGE initially empty
domain MSG·TYPE ≡ {advertisement, search, request, response,
                  revocation, dhcpOffer, dhcpDiscover}
```

A message uniquely identifies a sender, a receiver, a message type, and the actual message content. The content can be any finite representation of information to be transferred from a sender to a receiver.

```
sndr : MESSAGE → ADDRESS
rcvr : MESSAGE → ADDRESS
type : MESSAGE → MSG·TYPE
data : MESSAGE → DATA
```

3.2.3 Sending and Receiving Messages

An application is running on some host connected to one or more local networks. The operation of sending a message as well as the delivery of a message both require some form of direct interaction between this host and one of its local networks. We can assume here that the network is uniquely determined by the application. Abstractly, this relation is expressed using a unary dynamic function *network* defined on applications.

```
network : APPLICATION → COMMUNICATOR
```

Local Mailboxes. An agent has a local mailbox for storing messages until these messages will be processed. According to this view, the mailbox of a network agent represents the set of messages that are currently in transit on the related network. The mailbox of an application represents its local input port as identified by the respective port number for this application.

```
mailbox : AGENT → Set of MESSAGE initially empty
```

Message Output. We introduce the output operation defined below for applications to send a message over a local network. Here *me* refers to the application sending the message. The **extend** operation below creates a new message object. The effect of the send operation is that the message is put into the mailbox of the network agent.

```
Output(r:ADDRESS,d:DATA,t:MSG·TYPE)=
  extend  MESSAGE  with  m
     sndr(m):=address(me)
     rcvr(m):=r
     data(m):=d
     type(m):=t
     mailbox(network(me)) := mailbox(network(me)) ∪ {m}
```

Multicasting and Broadcasting. Depending on the applied mechanism for message delivery, one typically classifies addressing as *unicasting*, *multicasting* or *broadcasting*. Conceptually multicasting can be viewed as a generalization of all other address forms [7]. According to this view, we associate with every address some collection of applications, called a *multicast group*. More specifically, the group associated with a unicast address consists of a single application. The group associated with a broadcast address consists of all applications on a given local network. Notice that multicast groups change dynamically as applications come and go. Our model distinguished two basically different multicast groups: *control points* and *devices*.

3.3 Abstract Protocol Model

In this section we define a high-level ASM model of the UPnP protocol. This ASM model is then refined into an executable AsmL model by adding details related to the object-oriented specification style of AsmL as will be described in Section 4.

3.3.1 Initial States

An initial state reflects the particular system configuration under consideration. As such it identifies some finite collection of a priori given agents, one for each control point, each device and each communicator. Depending on the type of an agent, it executes the program *RunControPoint, RunDevice*, or *RunNetwork.* The behavior of DHCP server agents is not explicitly defined in terms of a program; rather it is determined by the respective actions controlled by the external world.

```
domain PROGRAM ≡ {RunControlPoint, RunDevice, RunNetwork}
```

Integrity Constraint. In every state of an ASM run, including the initial state, the following property holds.

```
∀ x ∈ AGENT: program(x) =
  RunControlPoint, if x ∈ CONTROLPOINT
  RunDevice, if x ∈ DEVICE
  RunNetwork, if x ∈ COMMUNICATOR
```

Remark. In the model of the UPnP protocol, we abstract from any device specific properties but concentrate on those aspects of behavior that are common to all UPnP devices according to [1]. The resulting device model thus provides a basic description that may be extended by adding the device specifics depending on the particular type of device. Technically this is achieved through a parallel composition of the two device models, the basic one and the specific one.

3.3.2 Network Model

We assume a TCP/IP network using TCP and UDP as transport protocols for the transfer of messages between applications running on different machines. Recall that UDP uses the same unreliable datagram delivery semantics as IP [7]. As such it provides a connectionless, best-effort message delivery service, where messages may be lost, duplicated, delayed or delivered out of order without receiving any notification. Hence, it is in the responsibility of an application to tolerate this behavior.

Delivery and Routing. Collectively, the communicators solve the task of globally transferring messages between applications running on hosts connected to the network. Communicators thus imitate the behavior of IP routers, where we encode the topological information in a dynamic mapping called `routingTable`. Intuitively a `routingTable` of a given communicator maps a non-local addresses to the correct neighboring communicators. Notice that a multicast address may refer to several network communicators.

```
routingTable : COMMUNICATOR → Map of ADDRESS to COMMUNICATOR
```

Furthermore, communicators handle the delivery of messages to destinations on a given local network. That is, given the destination address of a message in conjunction with a local network, a (possibly empty) set of related destinations on this network must be identified. We therefore introduce a dynamic mapping of addresses called `addressTable`. Intuitively, `addressTable` is a mapping from addresses of multicast groups to addresses of related group members.

```
addressTable : COMMUNICATOR → Map of ADDRESS to Set of ADDRESS
```

Time to Live (TTL). To limit the maximum number of routers that a message can pass on its way from the sender host to a destination host, a *time-to-live* or TTL, is assigned when the message is created. Each router decrements the TTL by one until the message eventually reaches its final destination or will be discarded. UPnP defines the initial TTL to be 4.

```
ttl : MESSAGE → {0,1,2,3,4} initially 4
```

Message Transfer. The transfer of messages may be delayed in an unpredictable manner depending on resource limitations of the underlying physical network. Since we abstract here from lower level network layers, the decision whether a messages is ready to be delivered in a given state of the network is stated through an externally controlled predicate `ReadyToDeliver`. (Notice that for some UDP message `m` the condition `ReadyToDeliver(m)` may never hold, implying that the message effectively gets lost.)

```
monitored ReadyToDeliver : MESSAGE → BOOL initially false
```

Now, we can formalize the network behavior in terms of interacting communicators executing the below program. This program performs three basically different steps, namely: (1) limited broadcasting within the local network; (2) delivery of multicast messages on a local network; (3) routing of messages through a global network. To identify local networks, a unique *network identifier*, called `netid`, is associated with each communicator. The network identifier can be derived from an address by inspecting the network mask part of the address. In the program below *me* refers to a communicator.

```
RunCommunicator ≡
  choose msg ∈ mailbox(me): ReadyToDeliver(msg) do
    mailbox(me):= mailbox(me) - {msg}
    if rcvr(msg) = broadcast then
      forall a ∈ APPLICATION: network(a) = me do
        DeliverMessageToMailbox(msg,address(a),a)
    else
      forall adr ∈ addressTable(me)(rcvr(msg)) do
        if netid(adr)= netid(me)  then
          choose a ∈ APPLICATION: address(a) = adr do
            DeliverMessageToMailbox(msg,adr,a)
        else
          if ttl(msg) > 0 then
            let c = routingTable(me)(adr) in
              DeliverMessageToMailbox(msg,adr,c)
```

The operation of delivering a message to the mailbox of a given agent is defined below. Applications and communicators are treated uniformly. They are both agents that have a mailbox and the operation performed on this mailbox (i.e., inserting a copy of some message) does not depend on the particular type of agent.

```
DeliverMessageToMailbox(msg:MESSAGE,adr:ADDRESS,ag:AGENT) =
  extend MESSAGE with m
    sndr(m):= sndr(msg),
    rcvr(m):= adr,
    type(m):= type(msg),
    data(m):= data(msg), ttl(m) := ttl(m)-1
    mailbox(ag):= mailbox(ag) ∪ {m}
```

3.3.3 Device Model

In this section we define the device model as parallel composition of a number of synchronously operating ASM models, each of which runs a different protocol step. For the purpose of illustrating the device program, we focus here on *Addressing*, *Discovery* and *Control.*

Device Status. In a given system state, a UPnP device may or may not be connected to a network. Regarding the connection status of a device, there are basically three different situations:

- *inactive*: the device is currently not connected to a network;
- *alive*: the device is connected and will probably remain connected for some time;
- *byebye*: the device is connected but is about to be removed from the network.

The status of a device is affected by actions and events in the external world. Therefore, it may change in a basically unpredictable way, but one can assume that devices initially are not connected. To model the device status, we introduce an externally controlled dynamic function *status* defined on devices.

```
monitored status : DEVICE → { inactive, alive, byebye }
```

In the device program defined below, *me* refers to a device agent.

```
RunDevice =
  if status(me) ≠ inactive then
    RunAddressing
    RunDiscovery
    RunDescription
    RunControl
    RunEventing
    RunPresentation
```

Addressing. IP address management requires a DHCP server to uniquely assign an IP address whenever a new host (for which no IP address is specified manually) is added to a network [8]. A key idea behind DHCP is to enable communication between a DHCP server and the host's DHCP client using the hardware address of the host. That is, as reply to a DHCPDISCOVER message from a client, the server broadcasts a DHCPOFFER message identifying the IP address as well as the hardware address of the host. By checking the hardware address, the host identifies itself as receiver (and can thus install the IP address).

Alternatively, a host may obtain a *temporary* IP address through auto IP addressing in case that a DHCP server is not available (or reachable). This temporary address may then be used until a matching DHCPOFFER message is received. To distinguish various situations with regard to a device's address status, we use the following abbreviations for checking whether a message is an offer from a matching DHCP server offer.

```
DhcpOffer(m) ≡
    (type(m) = dhcpoffer and hwAddressEncodedIn(data(m)) = hwAddress(me))
```

We abstract here from the device specific algorithm used for auto IP addressing by making a non-deterministic choice. To check the result, we assume to have some externally controlled decision procedure as represented by the monitored predicate *ValidAutoIPAdr.* Without specifying any further details, we expect the resulting auto IP addresses to fulfill the given constraints.

```
monitored ValidAutoIPAdr : DEVICE × ADDRESS → BOOL
```

```
RunAddressing ≡
  if address(me) = thisDevice or AutoConfiguredAddress(me) then
```

```
    RunDHCPclient
  if address(me) = thisDevice and ¬DhcpOfferReceived then
    choose address ∈ ADDRESS: ValidAutoIPAdr(me,address) do
      address(me):= address
    AutoConfiguredAddress(me):= true
```

where

```
  DhcpOfferReceived ≡ ∃ m ∈ mailbox(me): DhcpOffer(m)
```

Even in case that a DHCPOFFER has been received, a host may continue to use a temporary IP address for some time until it eventually switches to the server assigned IP address. In [1], the condition that affects the switching of addresses is left abstract. Accordingly, we model this behavior by introducing an externally controlled predicate *SwitchAddressEvent* defined on devices.

```
monitored SwitchAddressEvent : DEVICE → BOOL initially false
```

When a DHCP client becomes activated, it generates an initial DHCPDISCOVER message immediately. To distinguish this situation from those where a timeout event triggers the generation of subsequent DHCPDISCOVER messages, we introduce a special flag.

```
IssueInitialDiscover : DEVICE → BOOL initially true

RunDHCPclient =
  choose m ∈ mailbox(me): DhcpOffer(m) do
    if SwitchAddressEvent (me) then
      address(me):= rcvr(m)
      AutoConfiguredIPAdr(me):= false
      AdvertiseNewAds(rcvr(m))
      RevokeOldAds(address(me))
  if ¬DhcpOfferReceived then
    if Timeout(me,dhcpClientTimer) or IssueInitialDiscover(me) then
      Output(255.255.255.255,dhcpdiscover,hwAddress(me))
      SetTimer(me,dhcpClientTimer)
      IssueInitialDiscover(me):= false
where
  DhcpOfferReceived ≡ exists m ∈ mailbox(me): DhcpOffer(m)
```

After switching to a new address, the device must update and reissue its advertisements as well as revoke the old advertisements. These operations are further specified in the AsmL model.

Discovery. The discovery part of the UPnP protocol is based on UDP. Since messages may get lost, devices inform control points about their presence and the services they offer reissuing their advertisements on a regular basis. Additionally, a device replies to service requests from control points in case that a request matches with a service offered by the device, as indicated by the following predicate.

```
MatchingServiceRequest : SERVICE * MESSAGE → BOOL

RunDiscovery ≡
  NotifyAdsStatus
  RespondToSearch

NotifyAdsStatus ≡
  if address(me) ≠ thisDevice and Timeout(me,discoveryTimer) then
```

```
    SetTimer(me,discoveryTimer)
    if status(me) = alive then
      NotifyDeviceAvailable(controlPoints)
    if status(me) = byebye then
      NotifyDeviceUnavailable(controlPoints)
      status(me):= inactive
```

We identify the services associated with a root device or one of its embedded devices using a static function *srvcs* defined on devices.

```
srvcs : DEVICE → Set of SERVICE

RespondToSearch ≡
  choose m ∈ mailbox(me) : type(m) = search do
    mailbox(me)(m)  := false
    if exists s ∈ srvcs(me): MatchingServiceRequest(s,m) then
      NotifyDeviceAvailable(sndr(m))
```

The control part of a device is responsible for invoking the pending requests and generating the responses to those requests. The requests are handled one at a time. Each request message is assumed to have a data part that is a map identifying a service (`Service`), an action (`Action`) and arguments (`Arguments`) for that action. A response message is sent back to the sender of the request message. The data of the response message records the result of the action call. See the AsmL model for further details.

```
RunControl ≡
  if not address(me)=thisDevice then
    choose m ∈ mailbox(me) : type(m) = request and
                            data(m)(Service) ∈ srvcs(me) do
      let res = Invoke(data(m)(Service),data(m)(Action),data(m)(Arguments))
      mailbox(me)(m) := false
      Output(address(me), sndr(m), {Result mapsto res}, response)
```

4 Executable Protocol Model and GUI

The protocol model defined in the previous section can be refined into an executable AsmL program. The AsmL model is written in an object-oriented style. This is for example reflected by the fact that the identity of agents is mostly not explicitly mentioned but is implicitly given by the identity of the object (*me*). The object model was illustrated above with a UML class diagram. The intension is that the AsmL model is self-explanatory. Many of the comments regarding the logical structure of the model were already given above and are not repeated here.

The rest of the section is structured as follows. We start by mentioning the GUI. The top level loop that drives the model, as well as the methods defining the interaction between the model and the environment, are part of the interface definition to the GUI. Next we define the universes of the different kind of entities that we need. We then describe the communication model and the message structure needed to support it. Finally we define the AsmL models for communicators, control points and devices, respectively. The last three subsections constitute the main part of the overall AsmL model.

4.1 Gaphical User Interface

It is important to have a GUI that allows you to visualize the state in a way that is close to the intuitive understanding, and that allows you to interact with the AsmL model.

The following figure shows a snapshot of the simulator in a setup with two separate network communicators, one for devices and one for control points. There is a DHCP server whose behavior is completely controlled by the external world through the GUI. There is one control point and one device. The control point is receiving advertisements and revocations (of old advertisements) from the device (some of the advertisements are still in transfer), as a result of the device having received a new IP address from the DHCP server.

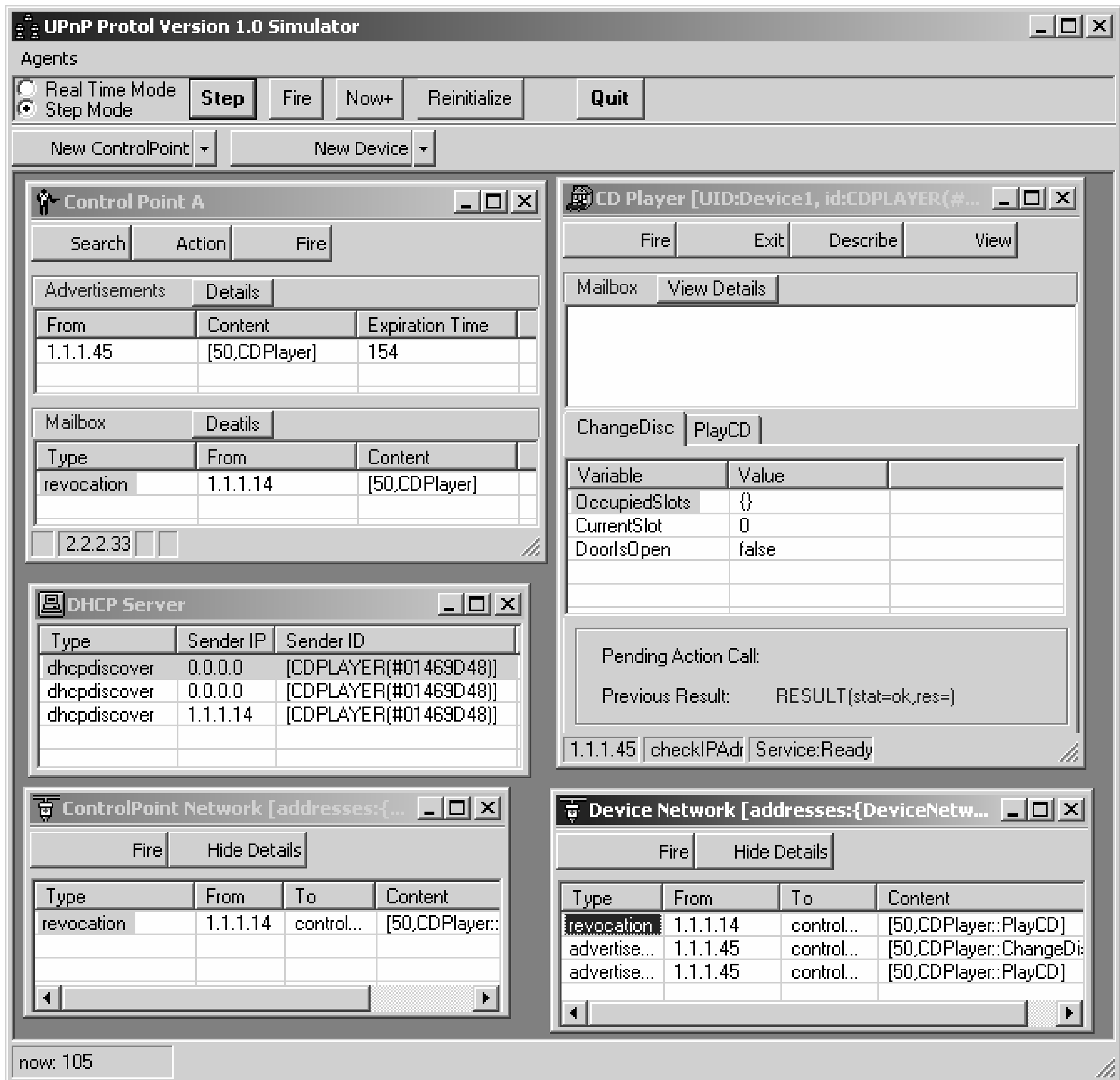

Figure 5: Snapshot of the UPnP Simulation tool.

In order to interact with the model using a GUI (as a client) we encapsulate the model in a COM component (acting as a server) that exposes the methods needed to interact with the model through an interface. The COM server is named `UPnPModelServer`.

```
import COM
serverName = "UPnPModelServer"
```

We declare an interface called `IServer` (extension with the builtin interface `IDispatch` is needed for automation support).

```
interface IServer extends IDispatch
```

The first method that the user of this server must do is to invoke the `Initialize` method. All `IServer` methods are declared and implemented in Appendix II.

4.2 Executable Protocol Model

```
var CONTROLPOINTs as Set[CONTROLPOINT] = {}
var DEVICEs as Set[DEVICE] = {}
var DHCPSERVERs as Set[DHCPSERVER] = {}

APPLICATIONs() as Set[APPLICATION] =
  {c as APPLICATION | c in CONTROLPOINTs}
  union {d as APPLICATION | d in DEVICEs}
  union {s as APPLICATION | s in DHCPSERVERs}

var COMMUNICATORs as Set[COMMUNICATOR] = {}
```

```
var now as Integer = 0
```

In order to run the programs of all the agents in AsmL, we introduce the following top-level rule. The executions of several agents are generally simulated by *interleaving*. Here we may in fact run them all *simultaneously* because they do not share variables, all communication between agents takes place via message passing.

```
RunUPnP() =
  forall c in CONTROLPOINTs do
    c.RunControlPoint()
  forall d in DEVICEs do
    d.RunDevice()
  forall n in COMMUNICATORs do
    n.RunNetwork()
```

Timers

```
enum TIMERTYPE
  dhcpClientTimer
  discoveryTimer

Timeout(d as DEVICE, t as TIMERTYPE) as Boolean =
  now >= d.time(t)
```

```
SetTimer(d as DEVICE, t as TIMERTYPE) =
  d.time(t):= now + d.duration(t)
```

4.3 *Communication Model*

In AsmL the host's object id is used as its hardware address.

```
hwAdr(h as APPLICATION) as String = asString(h)
```

4.3.1 IP Address Space

```
structure ADDRESS
  mask as String
  adr as String
  port as Integer
  asString() as String = adr

thisDevice = ADDRESS("","0.0.0.0",0)
UnspecifiedIPAdr(a as ADDRESS) as Boolean = (a = thisDevice)
```

4.3.1.1 Multicast and Broadcast Addresses

We distinguish two different multicast groups: devices and control points. Each multicast group is uniquely identified by a distinguished IP address.

```
controlPoints = ADDRESS(controlPointNetId,"2.2.2.255",0)
devices       = ADDRESS(deviceNetId,"1.1.1.255",0)
```

The distinguished IP address 255.255.255.255 is used for broadcast.

```
broadcast     = ADDRESS("","255.255.255.255",0)
```

4.3.2 Representation of Messages

```
var MESSAGEs as Set[MESSAGE] = {}

enum MSGTYPE
  advertisement
  search
  request
  response
  revocation
  dhcpoffer
  dhcpdiscover
```

Depending on the type of the message the data may include different fields of information each encoded as a string. The content is therefore a map from field identifiers to their respective values. The number of fields is determined by the type of the message.

```
enum FIELD
```

```
  Device
  Service
  Action
  Arguments
  Lifetime
  HardwareAddress
  NewAddress
  SearchPattern
  Result

structure DATA
  f as FIELD -> String
  service()  as String  =  f(Service)
  action()   as String  =  f(Action)
  args()     as String  =  f(Arguments)
  duration() as Integer =  Integer.fromString(f(Lifetime))
  hwAdr()    as String  =  f(HardwareAddress)
  newAdr()   as ADDRESS =  ADDRESS(deviceNetId,f(NewAddress),0)
  search()   as String  =  f(SearchPattern)

emptycontent as DATA = DATA({|->})

class MESSAGE
  sndr as ADDRESS
  rcvr as ADDRESS
  data as DATA
  tYPE as MSGTYPE
  var time as Integer
```

4.3.3 Sending and Receiving Messages

The underlying communication protocol is based on an asynchronous communication model. For simplicity, we assume unbounded communication bandwidth. Control points as well as devices may send and receive any finite number of messages at a time.

4.3.3.1 Agents

Each agent has a private mailbox into which messages can be inserted by other agents.

```
class AGENT()
  var mailbox as Set[MESSAGE] = {}
```

4.3.3.2 Applications

An application is an agent with a unique IP address and belongs to a certain network. It outputs its messages to the local network it belongs to.

```
class APPLICATION(a as ADDRESS, n as COMMUNICATOR) extends AGENT()
  var adr as ADDRESS       = a
  var network as COMMUNICATOR = n
```

```
  Output(s as ADDRESS, r as ADDRESS, c as DATA, t as MSGTYPE) =
    let m = new MESSAGE(s, r, c, t, now)
    MESSAGEs(m) := true
    network.mailbox(m) := true
```

4.4 Network Model

Each communicator has a netid, an address table and a routing table.

```
class COMMUNICATOR(t as String,
                   atbl as ADDRESS -> Set[ADDRESS],
                   rtbl as ADDRESS -> COMMUNICATOR) extends AGENT()
  netid as String                             = t
  addressTable as ADDRESS -> Set[ADDRESS]      = atbl
  var routingTable as ADDRESS -> COMMUNICATOR = rtbl
```

4.4.1 RunNetwork

```
class COMMUNICATOR...
  RunNetwork() =
    choose msg in mailbox do
      mailbox(msg) := false

      if msg.rcvr = broadcast then
        // limited broadcast to all local applications
        forall a in APPLICATIONs() where a.network = me do
          DeliverMessageToMailbox(msg,broadcast,a)

      else
        if addressTable(msg.rcvr) <> undef then
          forall adr in addressTable(msg.rcvr) do
            //if the address is local
            if adr.mask = netid then
              choose a in APPLICATIONs() where a.adr = adr do
                DeliverMessageToMailbox(msg,adr,a)
            else
              //obs! ttl is not implemented
              if routingTable(adr) <> undef then
                DeliverMessageToMailbox(msg,adr,routingTable(adr))

DeliverMessageToMailbox(m as MESSAGE, adr as ADDRESS, ag as AGENT) =
  let msg = new MESSAGE(m.sndr, m.rcvr, m.data, m.tYPE, m.time)
  MESSAGEs(msg)   := true
  ag.mailbox(msg) := true
```

4.4.2 DHCP Server

The behavior of dhcp servers is unspecified.

```
class DHCPSERVER(adr' as ADDRESS, network' as COMMUNICATOR)
  extends APPLICATION(adr', network')
```

4.5 Control Point Model

Control points may invoke actions. Each action request refers to a particular target device via its address (dadr), and contains an action call with a given name (actn) and arguments (args) targeted to a specific service (srvc) within that device.

```
structure ACTIONREQUEST
  dadr as ADDRESS
  srvc as String
  actn as String
  args as String

NoAction = ACTIONREQUEST(thisDevice,"","","")
```

A control point is a host that has a fixed type and UID.

```
class CONTROLPOINT(adr' as ADDRESS,
                   network' as COMMUNICATOR,
                   type' as String,
                   uid' as String) extends APPLICATION(adr',network')
  tYPE as String   = type'
  uid as String    = uid'
```

In a given state, we associate with each control point a (possibly empty) set of advertisements. Conceptually, advertisements are messages of type *advertisement*.

```
class CONTROLPOINT...
  var ads as Set[MESSAGE]       = {}
```

Run the control point.

```
class CONTROLPOINT...
  RunControlPoint() =
    SearchForDevices()
    ControlDevices()
    UpdateAds()
    EmptyMailbox()

  EmptyMailbox() =
    forall m in mailbox do
      mailbox(m) := false
```

4.5.1 Search for Devices

When a control point is added to the network, the UPnP discovery protocol allows that control point to search for devices of interest on the network. To specify the devices of interest for a given control point, a monitored function *searchPattern* yields a corresponding search pattern. The search pattern may either be a UID or a type of a device. We assume here that a search for devices may be invoked at any time.

```
class CONTROLPOINT...
  var searchPattern as String   = ""
  SearchForDevices() =
    if searchPattern <> "" then
      Output(adr,devices,DATA({SearchPattern |-> searchPattern}), search)
      searchPattern := ""
```

4.5.2 Processing Ads

```
class CONTROLPOINT...
  UpdateAds() =
    IncludeNewAds()
    RemoveExpiredAds()
    RemoveRevokedAds()

  IncludeNewAds() =
    forall m in mailbox where m.tYPE = advertisement do
      m.time     := now + m.data.duration() // set expiration time
      ads(m)     := true                    // save the ad

  RemoveExpiredAds() =
    forall ad in ads where ad.time <= now do
      ads(ad):= false

  RemoveRevokedAds() =
    forall m in mailbox where m.tYPE = revocation do
      forall ad in ads where ad.data = m.data and ad.sndr = m.sndr do
        ads(ad) := false
```

4.5.3 Controlling Devices

Each control point has a monitored function action that may contain a request to be issued by the control point.

```
class CONTROLPOINT...
  var action as ACTIONREQUEST = NoAction
  ControlDevices() =
    InvokeAction()
    ProcessResponse()

  InvokeAction() =
    if action <> NoAction then
      Output(adr,action.dadr,DATA({Service   |-> action.srvc,
                                   Action    |-> action.actn,
                                   Arguments |-> action.args}), request)
      action := NoAction

  ProcessResponse() =
    forall m in mailbox where m.tYPE = response do
```

```
      PrintResponse(me, m)    // external function
```

4.6 Device Model

The AsmL model for UPnP devices formalizes device behavior through the program of device agents. In a given machine state, each device agent executes the rule RunUPnPDevice defined below. The status of a device is either alive, byebye or inactive. It is given by the shared function `status`.

```
enum DEVICESTATUS
  alive
  byebye
  inactive
```

```
class DEVICE(adr' as ADDRESS,
             network' as COMMUNICATOR,
             type' as String,
             uid' as String,
             srvcs' as Set[SERVICE],
             ads' as Set[DATA],
             duration' as TIMERTYPE -> Integer,
             time' as TIMERTYPE -> Integer) extends APPLICATION(adr',network')
  tYPE as String                      = type'
  uid as String                       = uid'
  var srvcs as Set[SERVICE]           = srvcs'
  ads as Set[DATA]                 = ads'
  duration as TIMERTYPE -> Integer    = duration'
  var time as TIMERTYPE -> Integer    = time'
  var status as DEVICESTATUS = alive

  RunDevice() =
    if status <> inactive then
      RunAddressing()
      RunDiscovery()
      RunDescription()
      RunControl()
      RunPresentation()
      RunEventing()
```

4.6.1 Addressing

```
class DEVICE...
  var AutoConfiguredIPAdr as Boolean = false
  var ContinueAutoIP as Boolean = false
  SupportsAutoIPAdr as Boolean = true

  RunAddressing() =
    if UnspecifiedIPAdr(adr) or AutoConfiguredIPAdr then
```

```
      RunDHCPclient()                               //DHCP
    if UnspecifiedIPAdr(adr) and not DhcpOfferReceived() then
      if AutoIPEnabled() then RunAutoIPAddressing() //AutoIP

  AutoIPEnabled() as Boolean =
    SupportsAutoIPAdr and
    (Timeout(me,dhcpClientTimer) or ContinueAutoIP)

  DhcpOfferReceived() as Boolean =
    exists m in mailbox where m.tYPE = dhcpoffer and m.data.hwAdr() = hwAdr(me)
```

```
class DEVICE...
  var candidateAdr as ADDRESS = thisDevice
```

4.6.1.1 Dynamic Host Configuration Protocol

```
class DEVICE...
  var SwitchIPAdrEvent as Boolean = true
  var IssueInitialDiscover as Boolean = true
  RunDHCPclient() =
    if DhcpOfferReceived() then
      if SwitchIPAdrEvent then
        choose m in mailbox where m.tYPE = dhcpoffer and
                                  m.data.hwAdr() = hwAdr(me) do
          let newAdr = m.data.newAdr()
          mailbox(m)          := false
          MESSAGEs(m)         := false
          adr                 := newAdr
          AutoConfiguredIPAdr := false
          AdvertiseNewAds(newAdr)
          RevokeOldAds(adr)

    elseif Timeout(me,dhcpClientTimer) or
            IssueInitialDiscover then
      //provide the hardware address of the device in the message data
      Output(adr, broadcast,DATA({HardwareAddress |-> hwAdr(me)}),dhcpdiscover)
      SetTimer(me,dhcpClientTimer)
      IssueInitialDiscover := false

  RevokeOldAds(a as ADDRESS) =
    if not UnspecifiedIPAdr(adr) then
      forall ad in ads do
        Output(a, controlPoints, ad, revocation)

  AdvertiseNewAds(a as ADDRESS) =
    forall ad in ads do
      Output(a, controlPoints, ad, advertisement)
```

4.6.1.2 Auto-IP Addressing

```
class DEVICE...
  var CandidateAdrIsValid as Boolean = false
  var mode as AUTOIPMODE = chooseIPAdr

  RunAutoIPAddressing() =
    match mode with
      chooseIPAdr : ChooseIPAdr()
                    ContinueAutoIP := true
      probe       : Probe()
      checkIPAdr  : CheckIPAdr()

  ChooseIPAdr() =
    candidateAdr := guessAutoIPAdr(me)
    mode := probe

  Probe() =
    CandidateAdrIsValid :=
      not(exists h in APPLICATIONs() where h.adr = candidateAdr)
    mode := checkIPAdr

  CheckIPAdr() =
    if CandidateAdrIsValid then
      adr                 := candidateAdr
      AutoConfiguredIPAdr := true
      ContinueAutoIP     := false
    else
      mode := chooseIPAdr
```

4.6.2 Discovery

```
class DEVICE...
  RunDiscovery() =
    if not UnspecifiedIPAdr(adr) then
      RespondToSearch()
      if Timeout(me, discoveryTimer) then
        match status with
          alive:
            SetTimer(me, discoveryTimer)
            NotifyDeviceAvailable(controlPoints)
          byebye:
            NotifyDeviceUnavailable()
            status := inactive

  NotifyDeviceAvailable(rcvr as ADDRESS) =
```

```
    forall a in ads do
      Output(adr, rcvr, a, advertisement)

  NotifyDeviceUnavailable() =
    forall a in ads do
      Output(adr, controlPoints, a, revocation)

  RespondToSearch() =
    if (exists m in mailbox where m.tYPE = search) then
      choose m in mailbox where m.tYPE = search do
        mailbox(m)  := false
        MESSAGEs(m) := false
        if SearchMatches(m) and status=alive then
          NotifyDeviceAvailable(m.sndr)

  SearchMatches(m as MESSAGE) as Boolean =
    (m.data.search() = tYPE or m.data.search() = uid)
```

4.6.3 Description

```
class DEVICE...
  RunDescription() = skip
```

4.6.4 Presentation

```
class DEVICE...
  RunPresentation() = skip
```

4.6.5 Eventing

```
class DEVICE...
  RunEventing() = skip
```

4.6.6 Control

```
class DEVICE...
  RunControl() =
    if not UnspecifiedIPAdr(adr) then
      choose m in mailbox where m.tYPE=request and
          (exists s in srvcs where m.data.service()=s.GetId()) do
        let s = unique s in srvcs where m.data.service()=s.GetId()
        let res = s.Invoke(ACTIONCALL(m.data.action(), m.data.args()))
        mailbox(m) := false
        Output(adr, m.sndr, DATA({Result |-> res.asString()}), response)

      //clean up the mailbox
      forall m in mailbox where UnwantedMessage(m) do
        mailbox(m) := false
```

```
  UnwantedMessage(m as MESSAGE) as Boolean =
    m.tYPE = dhcpdiscover or
    (m.tYPE = dhcpoffer and not AutoConfiguredIPAdr
     and not UnspecifiedIPAdr(adr))

enum AUTOIPMODE
  chooseIPAdr
  probe
  checkIPAdr
```

4.6.7 SERVICE Interface

Each service implements the SERVICE interface that allows the device to get the id of this service and to invoke an action on this service.

```
structure ACTIONCALL
  name as String      //name of the action
  args as String      //arguments ..

enum RESULTSTATUS
  ok
  err

structure RESULT
  stat as RESULTSTATUS  //normal result or an error tag
  res as String         //the result value

interface SERVICE
  GetId() as String
  Invoke(actn as ACTIONCALL) as RESULT
```

5 Conclusions

We construct a high-level Abstract State Machine model for Universal Plug and Play. The model is based on the ASM paradigm for distributed systems with real-time constraints and is executable in principle. For practical execution, we use AsmL, the Abstract state machine Language, developed at Microsoft Research and integrated with Visual Studio and COM. This gives us an AsmL model, a refined version of the ASM model. The third part of this project is a GUI by means of which the runs of the AsmL model are controlled and inspected at various levels of detail as required for e.g. simulation and conformance testing.

While the ASM approach is very general, the domain of communication software is important enough to command a particular attention. There is a stream of ASM work specific to this domain; a good example is the SDL paper [14]. The current paper is another example. The common method in the stream is to turn, systematically and gradually, the given informal descriptions of complex distributed systems, like the UPnP standard [1], into high-level executable models. Our ASM model of the UPnP protocol is on the level of abstraction of the informal description. The basic objects of the protocol are present in the model, and players of the protocol become agents of the model. Moreover, the ASM model is component-based so that it splits into

three separate submodels (for control points, devices and the communication network respectively) with clearly identified interfaces.

Even though the current version of our UPnP model does not cover all protocol steps, the resulting formalization captures all the significant aspects of UPnP: concurrency, communication and timing. The abstract operational view of ASMs thereby allows a seamless integration of control and data-oriented aspects of behavior specifications. Moreover, we combine synchronous and asynchronous execution paradigms, associated with the application programs and the communication network respectively, in one common model. In that sense, there are no conceptual difficulties to include the missing details (protocol steps) as they all rely on the same architectural model. This becomes obvious by inspecting and running the executable model in [15].

Acknowledgements

We thank Jeffrey Schlimmer from the UPnP group at Microsoft for inspiring discussions and helpful background information about the development and standardization of UPnP. We thank Colin Campbell for inspiring discussions and cooperation on modeling of the individual UPnP devices.

References

1. UPnP Device Architecture V1.0. *Microsoft Universal Plug and Play Summit, Seattle 2000*, Microsoft Corporation, Jan. 2000.
2. Universal Plug and Play Forum. Official web site of the UPnP Forum. URL: www.upnp.org.
3. W. Grieskamp, Y. Gurevich, W. Schulte and M. Veanes. Testing with Abstract State Machines. In *Proc. ASM'2001.*
4. Y. Gurevich. Evolving Algebras 1993: Lipari Guide, *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, 9-36.
5. Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms", ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, 77-111.
6. W. Schulte et al. AsmL Version 1.5, Internal Report, Microsoft Research, Foundations of Software Engineering, Redmond, WA, April 2001.
7. D. E. Comer. Internetworking with TCP/IP, *Principles, Protocols, and Architectures*. Prentice Hall, 2000.
8. R. Droms and T. Lemon. The DHCP Handbook, *Understanding, Developing, and Managing Automated Configuration Services*. Macmillan Technical Publishing, 1999.
9. Microsoft Research, Foundations of Software Engineering, Redmond, USA, http://research.microsoft.com/foundations.
10. Abstract State Machines, http://www.eecs.umich.edu/gasm/.
11. ITU-T Recommendation Z.100: Languages for Telecommunications Applications - Specification and Description Language (SDL), Annex F: SDL Formal Semantics Definition, International Telecommunication Union, Geneva, 2000.
12. Y. Gurevich, W. Schulte and M. Veanes, Toward Industrial Strength Abstract State Machines, in *Proc. ASM'2001*, 2001.
13. Don Box, *Essential COM*, Addison-Wesley, Reading, MA, 1998.
14. R. Eschbach, U. Glässer, R. Gotzhein and A. Prinz. On the formal semantics of SDL-2000: a compilation approach based on an Abstract SDL Machine. In *Abstract State Machines - Theory and Applications*. Y. Gurevich, P.W. Kutter, M. Odersky and L. Thiele (Eds.), Lecture Notes in Computer Science, Vol. 1912, Springer-Verlag, 2000.
15. Y. Gurevich. The ASM Paradigm, in *Proc. ASM'2001*, 2001.

6 Appendix I: AsmL model of CD Player

The following is a sample CD Player device specification.

```
createCDPLAYER(netw as COMMUNICATOR) as DEVICE =
  let uid = "Device" + asString(size(DEVICEs)+1) //create a new UID
  let changer = new CHANGEDISC()
  let player = new PLAYCD()
  let ads =  { DATA({Lifetime |-> "50",
                     Device   |-> "CDPlayer"}),
               DATA({Lifetime |-> "50",
                     Service  |-> "CDPlayer::ChangeDisc"}),
               DATA({Lifetime |-> "50",
                     Service  |-> "CDPlayer::PlayCD"}) }
  let dmap = { dhcpClientTimer |-> 30,
               discoveryTimer  |-> 50 }
  let tmap = { dhcpClientTimer |-> now + 30,
               discoveryTimer  |-> now }
  let dev = new CDPLAYER(netw, uid, changer, player, ads, dmap, tmap)
  changer.device := dev
  player.device  := dev
  return(dev as DEVICE)

class CDPLAYER(netw' as COMMUNICATOR,
               duid' as String,
               changer' as CHANGEDISC,
               player' as PLAYCD,
               discoveryads as Set[DATA],
               dmap' as TIMERTYPE -> Integer,
               tmap' as TIMERTYPE -> Integer)
               extends DEVICE(thisDevice,
                              netw',
                              "CD Player",
                              duid',
                              {changer' as SERVICE,
                               player' as SERVICE},
                              discoveryads,
                              dmap',
                              tmap')
  changer as CHANGEDISC = changer'
  player  as PLAYCD     = player'
```

6.1 Extended SERVICE interface

```
interface ExtSERVICE extends SERVICE
```

Get UPnP related data.

```
interface ExtSERVICE...
  GetType() as String
  GetUID() as String
```

Get signature information (variable, constant, sensor names, action names).

```
interface ExtSERVICE...
  GetStateVars() as String
  GetStateConstants() as String
  GetStateSensors() as String
  GetActions() as Set[String]
```

Get values of state functions.

```
interface ExtSERVICE...
  GetStateVarValue(v as String) as String
  GetStateConstantValue(v as String) as String
```

Probe the sensors.

```
interface ExtSERVICE...
  IsStateSensorEnabled(v as String) as Integer
  GetStateSensorValue(v as String) as String
  SetSensorValue(sensor as String, val as String)
```

Get the value of the monitored function holding the pending action call.

```
interface ExtSERVICE...
  GetPendingActionCall() as String
```

Get high-level description of the state formulated in natural language.

```
interface ExtSERVICE...
  GetStateDescription() as String
```

Get the value of the last result.

```
interface ExtSERVICE...
  GetActionResult() as String
```

6.2 CHANGEDISC Service

```
class CHANGEDISC implements ExtSERVICE
  var device as CDPLAYER = undef
```

6.2.1 State

6.2.1.1 UPnP state variables

```
class CHANGEDISC...
  var OccupiedSlots as Set[Integer] = {}
  var CurrentSlot   as Integer   = 0
  var DoorIsOpen    as Boolean   = false
```

6.2.1.2 Sensors

```
class CHANGEDISC...
  var DoorIsStuck   as Boolean   = false
```

6.2.1.3 Constants

```
class CHANGEDISC...
  allSlots          as Set[Integer] = {0..4}
```

6.2.1.4 Action call and result

```
class CHANGEDISC...
  var action as ACTIONCALL = ACTIONCALL("","")
  var res    as RESULT     = RESULT(ok,"")
```

6.2.2 ExtSERVICE methods

6.2.2.1 UPnP data

```
class CHANGEDISC...
  GetType() as String = "ChangeDisc"
  GetUID()  as String = (device.uid + "::ChangeDisc")
```

6.2.2.2 Signature

```
class CHANGEDISC...
  GetStateVars() as String =
    "{OccupiedSlots,CurrentSlot,DoorIsOpen}"
  GetStateConstants() as String =
    "{allSlots}"
  GetStateSensors() as String =
    "{DoorIsStuck,trayHasDisc}"
  GetActions() as Set[String] =
    ({"AddDisc","NextDisc","PrevDisc","RandomDisc",
      "OpenDoor","CloseDoor","ToggleDoor",
      "HasTrayDisc","IsDoorOpen"})
```

6.2.2.3 Values

```
class CHANGEDISC...
  GetStateVarValue(v as String) as String =
    match v with
      "OccupiedSlots" : OccupiedSlots.asString()
      "CurrentSlot"   : CurrentSlot.asString()
      "DoorIsOpen"    : DoorIsOpen.asString()
  GetStateConstantValue(v as String) as String =
    match v with
      "allSlots"      : allSlots.asString()
```

6.2.2.4 Probe sensors

```
class CHANGEDISC...
  GetStateSensorValue(v as String) as String =
    match v with
      "DoorIsStuck"   : DoorIsStuck.asString()
      "trayHasDisc"   : trayHasDisc().asString()
  IsStateSensorEnabled(v as String) as Integer =
    match v with
      "DoorIsStuck"   : 1
```

```
      "trayHasDisc"    :
        if DoorIsOpen then
          1
        else
          0
  SetSensorValue(v as String, val as String) =
    match v with
      "DoorIsStuck"    :
        DoorIsStuck := (val = "true")
      "trayHasDisc"    :
        if DoorIsOpen then
          OccupiedSlots(CurrentSlot) := (val = "true")
```

6.2.2.5 GetPendingActionCall

```
class CHANGEDISC...
  GetPendingActionCall() as String =
    if action.name = "" then
      ""
    else
      action.name + "(" + action.args + ")"
```

6.2.2.6 GetStateDescription

```
class CHANGEDISC...
  GetStateDescription() as String =
    let s1 = if DoorIsOpen then
               "1. Door is open.\n"
             else
               "1. Door is closed\n"
    let s2 = if trayHasDisc() then
               "2. There is a CD on the tray.\n"
             else
               "2. There is no CD on the tray.\n"
    let s3 = if successors() = {} then
               "3. Current disc has no successors.\n"
             else
               "3. Current disc has a successor.\n"
    let s4 = if successors() = {} then
               "4. Current disc has no predecessors.\n"
             else
               "4. Current disc has a predecessor.\n"
    let s5 = if OccupiedSlots = {} then
               "5. The CD player is empty.\n"
             else
               "5. The CD player is not empty.\n"
    let s6 = if OccupiedSlots = allSlots then
               "6. The CD player is full.\n"
```

```
            else
              "6. The CD player has empty slots.\n"
    let s7 = if DoorIsStuck then
              "7. The door is stuck."
            else
              "7. The door is functional."
    return(s1 + s2 + s3 + s4 + s5 + s6 + s7)
```

6.2.2.7 GetActionResult

```
class CHANGEDISC...
  GetActionResult() as String =
    asString(res)
```

6.2.3 SERVICE methods

6.2.3.1 GetId

```
class CHANGEDISC...
  GetId()   as String = "ChangeDisc"
```

6.2.3.2 Invoke

```
class CHANGEDISC...
  Invoke(a as ACTIONCALL) as RESULT =
    machine
      action := a
      res    := RESULT(ok,"")
    step
      fireAction()
    step
      return res

  fireAction() =
    match action.name with
      "AddDisc"     : AddDisc()
      "NextDisc"    : NextDisc()
      "PrevDisc"    : PrevDisc()
      "RandomDisc" : RandomDisc()
      "OpenDoor"    : OpenDoor()
      "CloseDoor"   : CloseDoor()
      "ToggleDoor" : ToggleDoor()
      "HasTrayDisc": HasTrayDisc()
      "IsDoorOpen" : IsDoorOpen()
```

6.2.4 UPnP Action Definitions

6.2.4.1 Derived functions

```
class CHANGEDISC...
  trayHasDisc() as Boolean =
```

```
    CurrentSlot in OccupiedSlots

  successors()  as Set[Integer] =
    ({e | e in OccupiedSlots where e gt CurrentSlot})

  predecessors() as Set[Integer] =
    ({e | e in OccupiedSlots where e lt CurrentSlot})
```

6.2.4.2 Error conditions

```
class CHANGEDISC...
  UPnPerror(code as Integer) as Boolean =
    match code with
      701 : OccupiedSlots = {}
      702 : OccupiedSlots = allSlots
      704 : DoorIsStuck
      oth : false
```

6.2.4.3 AddDisc

```
class CHANGEDISC...
  AddDisc() =
    if not(UPnPerror(702) or (UPnPerror(704) and not DoorIsOpen)) then
      DoorIsOpen := true
      choose slot in allSlots difference OccupiedSlots do
        CurrentSlot := slot
    else
      if UPnPerror(704) and not(DoorIsOpen) then
        if UPnPerror(702) then
          res := RESULT(err,"702/704")
        else
          res := RESULT(err,"704")
      else
        res := RESULT(err,"702")
```

6.2.4.4 NextDisc

```
class CHANGEDISC...
  NextDisc() =
    if not(UPnPerror(701) or (UPnPerror(704) and DoorIsOpen)) then
      DoorIsOpen := false
      if successors() ne {} then
        CurrentSlot := setmin(successors())
      else
        CurrentSlot := setmin(OccupiedSlots)
    else
      if UPnPerror(704) and DoorIsOpen then
        if UPnPerror(701) then
          res := RESULT(err,"701/704")
        else
```

```
        res := RESULT(err,"704")
      else
        res := RESULT(err,"701")
```

6.2.4.5 PrevDisc

```
class CHANGEDISC...
  PrevDisc() =
    if not(UPnPerror(701) or (UPnPerror(704) and DoorIsOpen)) then
      DoorIsOpen := false
      if predecessors() ne {} then
        CurrentSlot := setmax(predecessors())
      else
        CurrentSlot := setmax(OccupiedSlots)
    else
      if UPnPerror(704) and DoorIsOpen then
        if UPnPerror(701) then
          res := RESULT(err,"701/704")
        else
          res := RESULT(err,"704")
      else
        res := RESULT(err,"701")
```

6.2.4.6 RandomDisc

```
class CHANGEDISC...
  RandomDisc() =
    if not (UPnPerror(701) or (UPnPerror(704) and DoorIsOpen)) then
      DoorIsOpen := false
      choose e in OccupiedSlots do
        CurrentSlot := e
    else
      if UPnPerror(704) and DoorIsOpen then
        if UPnPerror(701) then
          res := RESULT(err,"701/704")
        else
          res := RESULT(err,"704")
      else
        res := RESULT(err,"701")
```

6.2.4.7 OpenDoor

```
class CHANGEDISC...
  OpenDoor() =
    if not (UPnPerror(704) and not DoorIsOpen) then
      DoorIsOpen := true
    else
      res := RESULT(err,"704")
```

6.2.4.8 CloseDoor

```
class CHANGEDISC...
```

```
  CloseDoor() =
    if not (UPnPerror(704) and DoorIsOpen) then
      DoorIsOpen := false
    else
      res := RESULT(err,"704")
```

6.2.4.9 ToggleDoor

```
class CHANGEDISC...
  ToggleDoor() =
    if not UPnPerror(704) then
      DoorIsOpen := not DoorIsOpen
    else
      res := RESULT(err,"704")
```

6.2.4.10 HasTrayDisc

```
class CHANGEDISC...
  HasTrayDisc() =
    res:= RESULT(ok,trayHasDisc().asString())
```

6.2.4.11 IsDoorOpen

```
class CHANGEDISC...
  IsDoorOpen() =
    res:= RESULT(ok,DoorIsOpen.asString())
```

6.2.4.12 Helper functions

```
class CHANGEDISC...
setmax(s as Set[Integer]) as Integer =
  (unique e | e in s where (forall d in s holds e gte d))
setmin(s as Set[Integer]) as Integer =
  (unique e | e in s where (forall d in s holds e lte d))
```

6.3 PLAYCD Service

```
class PLAYCD implements ExtSERVICE
  var device as CDPLAYER = undef
```

6.3.1 State

6.3.1.1 UPnP state variables

```
class PLAYCD...
  var PlayMode    as String  = "Stopped"
  var PlayProgram as String  = "None"
  var TrackNumber as Integer = 1
  var TrackOffset as Integer = 1
```

6.3.1.2 Sensors

```
class PLAYCD...
  var DiscIsUnreadable as Boolean   = false
```

6.3.1.3 Constants

```
class PLAYCD...
  DiscTOC as Integer -> Integer = {1 |-> 10, 2 |-> 20, 3 |-> 20,
                                   4 |-> 20, 5 |-> 20}
```

6.3.1.4 Next action and last result

```
class PLAYCD...
  var action as ACTIONCALL    = ACTIONCALL("","")
  var res    as RESULT        = RESULT(ok,"")
```

6.3.2 ExtSERVICE

```
class PLAYCD...
  GetType() as String =
    "PlayCD"
  GetUID() as String =
    device.uid + "::PlayCD"
  GetStateVars() as String =
    "{PlayMode,PlayProgram,TrackNumber,TrackOffset}"
  GetStateConstants() as String =
    "{DiscTOC}"
  GetStateSensors() as String =
    "{DiscIsUnreadable}"
  GetActions() as Set[String] =
    {"Play"
    ,"Pause"
    ,"Stop"
    ,"SetPlayProgram"
    ,"SelectTrack"
    ,"NextTrack"
    ,"PrevTrack"}
  GetStateVarValue(v as String) as String =
    match v with
      "PlayMode"    : asString(PlayMode)
      "PlayProgram" : asString(PlayProgram)
      "TrackNumber" : asString(TrackNumber)
      "TrackOffset" : asString(TrackOffset)
      others        : ""
  GetStateConstantValue(v as String) as String =
    match v with
      "DiscTOC" : asString(DiscTOC)
      others    : ""
  GetStateSensorValue(v as String) as String =
    match v with
      "DiscIsUnreadable" : asString(DiscIsUnreadable)
      others             : ""
```

```
  IsStateSensorEnabled(v as String) as Integer =
    1
  GetPendingActionCall() as String =
    if action.name = "" then
      ""
    else
      action.name + "(" + action.args + ")"
  GetStateDescription() as String =
    "PlayCD state description not implemented"
  SetSensorValue(sensor as String, val as String) =
    match sensor with
      "DiscIsUnreadable"   : DiscIsUnreadable := (val = "true")
      others               : skip
```

6.3.2.1 GetActionResult

```
class PLAYCD...
  GetActionResult() as String =
    asString(res)
```

6.3.3 SERVICE

6.3.3.1 GetId

```
class PLAYCD...
  GetId()   as String = "PlayCD"
```

6.3.3.2 Invoke

```
class PLAYCD...
  Invoke(a as ACTIONCALL) as RESULT =
    machine
      action := a
      res    := RESULT(ok,"")
    step
      fireAction()
    step
      return res

  fireAction() =
    match action.name with
      "Play"            : Play()
      "Pause"           : Pause()
      "Stop"            : Stop()
      "SetPlayProgram"  : SetPlayProgram(action.args)
      "SelectTrack"     : SelectTrack(asInteger(action.args))
      "NextTrack"       : NextTrack()
      "PrevTrack"       : PrevTrack()
      others            : skip
```

6.3.4 UPnP Action Definitions

6.3.4.1 Derived functions

```
class PLAYCD...
  tracks() as Set[Integer] = dom(DiscTOC)

  discNumberofTracks() as Integer = size(tracks())

  trackDuration() as Integer =
    if TrackNumber = 0 then 0 else DiscTOC(TrackNumber)

  isValidTrack(i as Integer) as Boolean =
    (i > 0) and (i < discNumberofTracks())

  isLastTrack() as Boolean  =  (TrackNumber = discNumberofTracks())
  isFirstTrack() as Boolean =  (TrackNumber = 1)

  isRandom() as Boolean  =
    PlayProgram in {"ONCE_RANDOM", "REPEAT_RANDOM"}

  isRepeated() as Boolean  =
    PlayProgram in {"REPEAT_IN_ORDER", "REPEAT_RANDOM"}

  discHasTracks() as Boolean = discNumberofTracks() > 0
  discHasTooManyTracks() as Boolean = discNumberofTracks() > 255
```

6.3.4.2 Error conditions

```
class PLAYCD...
  UPnPerror(code as Integer) as Boolean =
    match code with
      501 : not(DiscIsUnreadable) and UPnPerror(701)
      701 : not(device.changer.trayHasDisc()) and
            not(device.changer.DoorIsOpen)
      703 : device.changer.DoorIsOpen
      711 : not(discHasTracks()) and
            DiscIsUnreadable and
            UPnPerror(701)
      712 : discHasTooManyTracks() and
            DiscIsUnreadable and
            UPnPerror(701)
      799 : exists e in {701,703,711,712} where UPnPerror(e)
```

6.3.4.3 Play

```
class PLAYCD...
  Play() =
    if not(UPnPerror(501) or UPnPerror(799)) then
      PlayMode := "Playing"
```

```
    else
      if UPnPerror(501) then
        if UPnPerror(799) then
          res := RESULT(err,"501/7??")
        else
          res := RESULT(err,"501")
      else
        res := RESULT(err,"7??")
```

6.3.4.4 Pause

```
class PLAYCD...
  Pause() =
    if not(UPnPerror(501) or UPnPerror(799)) then
      PlayMode := "Paused"
    else
      if UPnPerror(501) then
        if UPnPerror(799) then
          res := RESULT(err,"501/7??")
        else
          res := RESULT(err,"501")
      else
        res := RESULT(err,"7??")
```

6.3.4.5 Stop

```
class PLAYCD...
  Stop() =
    PlayMode    := "Stopped"
    TrackOffset := 0
    if device.changer.trayHasDisc() then
      TrackNumber := 1
    else
      TrackNumber := 0
```

6.3.4.6 SetPlayProgram

```
class PLAYCD...
  SetPlayProgram(pgm as String) = PlayProgram := pgm
```

6.3.4.7 SelectTrack

```
class PLAYCD...
  SelectTrack(newTrack as Integer) =
    if newTrack in tracks() and not(UPnPerror(799)) then
      TrackNumber := newTrack
      TrackOffset := 0
    else
      if UPnPerror(799) then res := RESULT(err,"7??")
```

6.3.4.8 NextTrack

```
class PLAYCD...
```

```
  NextTrack() =
    if not(UPnPerror(799)) then
      if isRandom() then
        choose t in tracks() do TrackNumber := t
      elseif isLastTrack() then
        TrackNumber := 1
      else
        TrackNumber := TrackNumber + 1
      TrackOffset := 0
    else
      res := RESULT(err,"7??")
```

6.3.4.9 PrevTrack

```
class PLAYCD...
  PrevTrack() =
    if not(UPnPerror(799)) then
      if isRandom() then
        choose t in tracks() do TrackNumber := t
      elseif isFirstTrack() then
        TrackNumber := discNumberofTracks()
      else
        TrackNumber := TrackNumber - 1
      TrackOffset := 0
    else
      res := RESULT(err,"7??")
```

7 Appendix II: IServer Interface

The following contains the declarations of all the `IServer` interface methods and the definition of the class `UPnPModelServer` implementing those methods. This interface is used by the GUI to visualize the AsmL state and to interact with the model.

7.1 IServer Declaration

```
interface IServer...
  shared guid as GUID = "3CB6F20B-4041-424C-A356-D48525A969ED"
```

7.1.1 General

```
interface IServer...
  Initialize()
  Reinitialize()
  Step()
  Create_NETWORK(info as String)        as String
  Create_CONTROLPOINT(info as String)   as String
  Create_DEVICE(info as String)         as String
```

7.1.2 NETWORK

```
interface IServer...
  NETWORK_LoseMessage(netw as String, msg as String)
  NETWORK_Terminate(netw as String)
  NETWORK_Step(netw as String)
  NETWORK_CollectMessage(netw as String)
  NETWORK_DeliverMessage(netw as String, msg as String)
  NETWORK_ReadMessagesInTransit(netw as String) as String
  NETWORK_GetMessageSender(netw as String, msg as String) as String
  NETWORK_GetMessageReceiver(netw as String, msg as String) as String
  NETWORK_GetMessageContent(netw as String, msg as String) as String
  NETWORK_GetMode(netw as String) as String
```

7.1.3 CONTROLPOINT

```
interface IServer...
  CONTROLPOINT_Search(ctrl as String, searchPattern as String)
  CONTROLPOINT_Terminate(ctrl as String)
  CONTROLPOINT_InvokeAction(ctrl as String,
                            srvc as String,
                            actn as String,
                            args as String)
  CONTROLPOINT_Discover(ctrl as String, what as String)
  CONTROLPOINT_GetOutBox(ctrl as String) as String
  CONTROLPOINT_GetInBox(ctrl as String) as String
  CONTROLPOINT_GetAdvertisements(ctrl as String) as String
  CONTROLPOINT_GetMode(ctrl as String) as String
  CONTROLPOINT_GetInMessage(ctrl as String, msg as String) as String
  CONTROLPOINT_DeleteInMessage(ctrl as String, msg as String)
  CONTROLPOINT_SaveInMessageInAds(ctrl as String, msg as String)
  CONTROLPOINT_GetOutMessage(ctrl as String, msg as String) as String
  CONTROLPOINT_GetAd(ctrl as String, msg as String) as String
  CONTROLPOINT_DeleteAd(ctrl as String, msg as String)
  CONTROLPOINT_InvokeServiceAction(ctrl as String,
                                   dadr as String,
                                   srvc as String,
                                   actn as String,
                                   args as String)
  CONTROLPOINT_GetUID(ctrl as String) as String
  CONTROLPOINT_GetIPADR(ctrl as String) as String
  CONTROLPOINT_Step(ctrl as String)
  CONTROLPOINT_GetPattern(ctrl as String) as String
  CONTROLPOINT_GetAction(ctrl as String) as String
```

7.1.4 DEVICE

```
interface IServer...
  DEVICE_SetSensor(dev as String, sensor as String, val as String)
  DEVICE_GetSensor(dev as String, sensor as String) as String
  DEVICE_Terminate(dev as String)
  DEVICE_Exit(dev as String)
  DEVICE_Step(dev as String)
  DEVICE_GetUUID(dev as String) as String
  DEVICE_GetType(dev as String) as String
  DEVICE_GetMode(dev as String) as String
  DEVICE_GetInBox(dev as String) as String
  DEVICE_GetInBoxMessage(dev as String, msg as String) as String
  DEVICE_GetOutBox(dev as String) as String
  DEVICE_GetOutBoxMessage(dev as String, msg as String) as String
  DEVICE_GetServices(dev as String) as String
  DEVICE_GetServiceUUID(dev as String, svc as String) as String
  DEVICE_GetServiceType(dev as String, svc as String) as String
  DEVICE_GetServiceMode(dev as String, svc as String) as String
  DEVICE_GetServiceStateVars(dev as String, svc as String) as String
  DEVICE_GetServiceStateVarValue(dev as String, svc as String,
                               v   as String) as String
  DEVICE_GetServiceSensors(dev as String,
                         svc as String) as String
  DEVICE_GetServiceSensorValue(dev as String,
                             svc as String,
                             sensor as String) as String
  DEVICE_IsServiceSensorEnabled(dev as String,
                              svc as String,
                              sensor as String) as Integer
  DEVICE_GetServicePendingActionCall(dev as String,
                                   svc as String) as String
  DEVICE_GetServiceActionResult(dev as String,
                              svc as String) as String
  DEVICE_GetServiceStateDescription(dev as String,
                                  svc as String) as String
  DEVICE_SetServiceSensorValue(dev as String,
                             svc as String,
                             sensor as String,
                             val as String)
  DEVICE_GetServiceStateConstants(dev as String,
                                svc as String) as String
  DEVICE_GetServiceStateConstantValue(dev as String,
                                    svc as String,
                                    cons as String) as String
```

```
  DEVICE_GetIPADR(dev as String) as String
  DEVICE_GetServiceActions(dev as String, svc as String) as String
  DEVICE_SetStatus(dev as String, stat as String)
```

7.1.5 Simulation Information

```
interface IServer...
  GetNETWORKTypes() as String
  GetDEVICETypes() as String
  GetCONTROLPOINTTypes() as String
  GetNrOfSteps() as Integer
```

7.1.6 MESSAGE

```
interface IServer...
  MESSAGE_Get(msg as String) as String
  MESSAGE_GetContent(msg as String) as String
  MESSAGE_GetHeader(msg as String) as String
  MESSAGE_GetReceiver(msg as String) as String
  MESSAGE_GetSender(msg as String) as String
  MESSAGE_GetTime(msg as String) as String
  MESSAGE_GetType(msg as String) as String
```

7.1.7 Time

```
interface IServer...
  SetNow(n as Integer)
  IncrNow(n as Integer)
  GetNow() as Integer
```

7.1.8 DHCP Server

```
interface IServer...
  DHCPServerReply(rcvr as String, id as String, adr as String)
  DHCPServerGetMailbox() as String
  DHCPServerDeleteMsg(msg as String)
```

7.1.9 Fire

```
interface IServer...
  Fire()
  NETWORK_Fire(netw as String)
  CONTROLPOINT_Fire(ctrl as String)
  DEVICE_Fire(dev as String)
//end interface IServer
```

7.1.10 Other

```
interface IServer...
  Get_Networks() as String
  NETWORK_GetType(netw as String) as String
```

```
  NETWORK_GetIP(netw as String) as String
```

7.1.11 Creation of Server

```
getCOMClasses() as Seq[GUID] =
  [GUID("6390E481-FD89-41C2-8552-EE96EF2918E6")]

createCOMInstance(clid as GUID) as IDispatch =
  new UPnPModelServer()
```

7.2 IServer Implementation

```
class UPnPModelServer implements IServer, AUTOMATION
```

7.2.1 Create_NETWORK

```
class UPnPModelServer...
  Create_NETWORK(v as String) as String =
    POPUP("Create_NETWORK is not implemented")
    return("")
```

7.2.2 Create_CONTROLPOINT

```
class UPnPModelServer...
  Create_CONTROLPOINT(v as String) as String =
    choose a in availableControlPointAddresses() do
      let ctrl = new CONTROLPOINT(a, controlPointNetwork, v, "")
      CONTROLPOINTs(ctrl) := true
      return(asString(ctrl))
    ifnone
      return("")
```

7.2.3 Create_DEVICE

Only devices of type *CD Player* are supported below, the argument v is ignored. Each device has `deviceNetwork` as its network communicator.

```
class UPnPModelServer...
  Create_DEVICE(v as String) as String =
    let dev = createCDPLAYER(deviceNetwork)
    DEVICEs(dev) := true
    return(asString(dev))
```

7.2.4 NETWORK

```
class UPnPModelServer...
  NETWORK_LoseMessage(netw as String, msg as String) =
    IdToNetwork(netw).mailbox(IdToMSG(msg)) := false
  NETWORK_Terminate(netw as String) =
    COMMUNICATORs(IdToNetwork(netw)) := false
  NETWORK_Step(netw as String) =
    try
```

```
      IdToNetwork(netw).RunNetwork()
      now := now + 1
    catch
      e as Object : POPUP("Exception in RunNetwork: " + e.asString())
  NETWORK_CollectMessage(netw as String) =
    POPUP("NETWORK_CollectMessage not implemented!")
  NETWORK_DeliverMessage(netw as String, msg as String) =
    POPUP("NETWORK_DeliverMessage not implemented")
    //IdToNetwork(netw).DeliverMessage(IdToMSG(msg))
  NETWORK_ReadMessagesInTransit(netw as String) as String =
    IdToNetwork(netw).mailbox.asString()
  NETWORK_GetMessageSender(netw as String, msg as String) as String =
    IdToMSG(msg).sndr.asString()
  NETWORK_GetMessageReceiver(netw as String,
                             msg as String) as String =
    IdToMSG(msg).rcvr.asString()
  NETWORK_GetMessageContent(netw as String, msg as String) as String =
    IdToMSG(msg).data.asString()
  NETWORK_GetMode(netw as String) as String =
    "active"
```

7.2.5 CONTROLPOINT

```
class UPnPModelServer...
  CONTROLPOINT_Search(ctrl as String, pattern as String) =
    POPUP("Not implemented, use 'CONTROLPOINT_Discover'")
  CONTROLPOINT_Terminate(ctrl as String) =
    CONTROLPOINTs(IdToControlPoint(ctrl)) := false
  CONTROLPOINT_InvokeAction(ctrl as String, srvc as String,
                            actn as String, args as String) =
    POPUP("Not implemented, use 'CONTROLPOINT_InvokeServiceAction'!")
  CONTROLPOINT_Discover(ctrl as String, pat as String) =
    IdToControlPoint(ctrl).searchPattern := pat
  CONTROLPOINT_GetOutBox(ctrl as String) as String =
    "{}"
  CONTROLPOINT_GetInBox(ctrl as String) as String =
    IdToControlPoint(ctrl).mailbox.asString()
  CONTROLPOINT_GetAdvertisements(ctrl as String) as String =
    IdToControlPoint(ctrl).ads.asString()
  CONTROLPOINT_GetMode(ctrl as String) as String =
    ""
  CONTROLPOINT_GetInMessage(ctrl as String, msg as String) as String =

      POPUP("CONTROLPOINT_GetInMessage not implemented!")
      return("")
  CONTROLPOINT_DeleteInMessage(ctrl as String, msg as String) =
```

```
    IdToControlPoint(ctrl).mailbox(IdToMSG(msg)) := false
  CONTROLPOINT_SaveInMessageInAds(ctrl as String, msg as String) =
    let c = IdToControlPoint(ctrl)
    let m = IdToMSG(msg)
    c.mailbox(m) := false
    c.ads(m)     := true
  CONTROLPOINT_GetOutMessage(ctrl as String,msg as String) as String =
     POPUP("CONTROLPOINT_GetOutMessage not implemented!")
     return("")
  CONTROLPOINT_GetAd(ctrl as String, msg as String) as String =
     POPUP("CONTROLPOINT_GetAd not implemented!")
     return("")
  CONTROLPOINT_DeleteAd(ctrl as String, msg as String) =
    IdToControlPoint(ctrl).ads(IdToMSG(msg)) := false
  CONTROLPOINT_InvokeServiceAction(ctrl as String,
                                   dadr as String,
                                   srvc as String,
                                   actn as String,
                                   args as String) =
    IdToControlPoint(ctrl).action :=
      ACTIONREQUEST(ADDRESS(deviceNetId,dadr,0), srvc, actn, args)
  CONTROLPOINT_GetUID(ctrl as String) as String =
    IdToControlPoint(ctrl).uid.asString()
  CONTROLPOINT_GetIPADR(ctrl as String) as String =
    IdToControlPoint(ctrl).adr.asString()
  CONTROLPOINT_Step(ctrl as String) =
    try
      IdToControlPoint(ctrl).RunControlPoint()
      now := now + 1
    catch
      e as Object :
        POPUP("Exception in RunControlPoint: " + e.asString())

  CONTROLPOINT_GetPattern(ctrl as String) as String =
    IdToControlPoint(ctrl).searchPattern.asString()
  CONTROLPOINT_GetAction(ctrl as String) as String =
    let a = IdToControlPoint(ctrl).action
    if a = NoAction then
      return("")
    else
      return(a.dadr.asString() + ":" + a.srvc + ":" +
            a.actn + "(" + a.args + ")")
```

7.2.6 DEVICE

```
class UPnPModelServer...
```

```
DEVICE_SetSensor(dev as String, sensor as String, val as String) =
  POPUP("DEVICE_SetSensor not implemented!")
DEVICE_GetSensor(dev as String, sensor as String) as String =
   POPUP("DEVICE_GetSensor not implemented!")
   return("device_sensor")
DEVICE_Terminate(dev as String) =
  DEVICEs(IdToDevice(dev)) := false
DEVICE_Exit(dev as String) =
  IdToDevice(dev).status   := byebye
DEVICE_Step(dev as String) =
  try
    IdToDevice(dev).RunDevice()
    now := now + 1
  catch
    e as Object : POPUP("Exception in RunDevice: " + e.asString())
DEVICE_GetUUID(dev as String) as String =
  IdToDevice(dev).uid
DEVICE_GetType(dev as String) as String =
  IdToDevice(dev).tYPE
DEVICE_GetMode(dev as String) as String =
  IdToDevice(dev).mode.asString()
DEVICE_GetInBox(devId as String) as String =
  IdToDevice(devId).mailbox.asString()
DEVICE_GetInBoxMessage(dev as String, msg as String) as String =
  MESSAGE_GetHeader(msg)
DEVICE_GetOutBox(devId as String) as String =
  "{}"
DEVICE_GetOutBoxMessage(dev as String, msg as String) as String =
  MESSAGE_GetHeader(msg)
DEVICE_GetServices(dev as String) as String =
  ({(srvc as ExtSERVICE).GetId() |
     srvc in IdToDevice(dev).srvcs}).asString()
DEVICE_GetServiceUUID(dev as String, svc as String) as String =
  IdToSrvc(IdToDevice(dev),svc).GetUID()
DEVICE_GetServiceType(dev as String, svc as String) as String =
  IdToSrvc(IdToDevice(dev),svc).GetType()
DEVICE_GetServiceMode(dev as String, svc as String) as String =
  ""
DEVICE_GetServiceStateVars(dev as String, svc as String) as String =
  IdToSrvc(IdToDevice(dev),svc).GetStateVars().asString()
DEVICE_GetServiceStateVarValue(dev as String,
                               svc as String,
                               v   as String) as String =
  IdToSrvc(IdToDevice(dev),svc).GetStateVarValue(v)
DEVICE_GetServiceSensors(dev as String,
```

```
                           svc as String) as String =
    IdToSrvc(IdToDevice(dev),svc).GetStateSensors().asString()
  DEVICE_GetServiceSensorValue(dev as String,
                               svc as String,
                               sensor as String) as String =
    IdToSrvc(IdToDevice(dev),svc).GetStateSensorValue(sensor)
  DEVICE_IsServiceSensorEnabled(dev as String,
                                svc as String,
                                sensor as String) as Integer =
    IdToSrvc(IdToDevice(dev),svc).IsStateSensorEnabled(sensor)
  DEVICE_GetServicePendingActionCall(dev as String,
                                     svc as String) as String =
    IdToSrvc(IdToDevice(dev),svc).GetPendingActionCall()
  DEVICE_GetServiceActionResult(dev as String,
                                svc as String) as String =
    IdToSrvc(IdToDevice(dev),svc).GetActionResult().asString()
  DEVICE_GetServiceStateDescription(dev as String,
                                    svc as String) as String =
    IdToSrvc(IdToDevice(dev),svc).GetStateDescription()
  DEVICE_SetServiceSensorValue(dev as String,
                               svc as String,
                               sensor as String,
                               val as String) =
    IdToSrvc(IdToDevice(dev),svc).SetSensorValue(sensor, val)
  DEVICE_GetServiceStateConstants(dev as String,
                                  svc as String) as String =
    IdToSrvc(IdToDevice(dev),svc).GetStateConstants().asString()
  DEVICE_GetServiceStateConstantValue(dev as String,
                                      svc as String,
                                      cons as String) as String =
    IdToSrvc(IdToDevice(dev),svc).GetStateConstantValue(cons)
  DEVICE_GetIPADR(dev as String) as String =
    IdToDevice(dev).adr.asString()
  DEVICE_GetServiceActions(dev as String, svc as String) as String =
    IdToSrvc(IdToDevice(dev),svc).GetActions().asString()
  DEVICE_SetStatus(dev as String, stat as String) =
    match stat with
      "alive"    : IdToDevice(dev).status := alive
      "byebye"   : IdToDevice(dev).status := byebye
      "inactive" : IdToDevice(dev).status := inactive
```

7.2.7 Simulation Parameters

```
class UPnPModelServer...
  GetNETWORKTypes() as String =
    "{Device Network,ControlPoint Network}"
```

```
  GetDEVICETypes() as String  = "{CD Player}"
  GetCONTROLPOINTTypes() as String =
    "{Control Point A,Control Point B}"
  GetNrOfSteps() as Integer =
     POPUP("GetNrOfSteps not implemented!")
     return(0)
```

7.2.8 MESSAGE

```
class UPnPModelServer...
  MESSAGE_Get(msg as String) as String =
    let m = IdToMSG(msg)
    return("From:" + m.sndr.asString() + "," +
          "To:"   + m.rcvr.asString() + "," +
          "Type:" + m.tYPE.asString() + "," +
          "Content:" + m.data.asString() + "," +
          "Time:" + m.time.asString())
  MESSAGE_GetContent(msg as String) as String =
    IdToMSG(msg).data.f.asString()
  MESSAGE_GetHeader(msg as String) as String =
    let m = IdToMSG(msg)
    return ("From:" + m.sndr.asString() + "," +
           "To:"   + m.rcvr.asString() + "," +
           "Type:" + m.tYPE.asString())
  MESSAGE_GetReceiver(msg as String) as String =
    IdToMSG(msg).rcvr.asString()
  MESSAGE_GetSender(msg as String) as String =
    IdToMSG(msg).sndr.asString()
  MESSAGE_GetTime(msg as String) as String =
    IdToMSG(msg).time.asString()
  MESSAGE_GetType(msg as String) as String =
    IdToMSG(msg).tYPE.asString()
```

7.2.9 Time

```
class UPnPModelServer...
  SetNow(n as Integer) =
    now := n
  IncrNow(n as Integer) =
    now := now + n
  GetNow() as Integer =
    now
```

7.2.10 DHCP Server

```
class UPnPModelServer...
  DHCPServerReply(temp as String, id as String, newAdr as String) =
    let msg = IdToMSG(id)
```

```
    dhcpserver.mailbox(msg) := false
    MESSAGEs(msg) := false
    if msg.sndr <> thisDevice then
      dhcpserver.Output(dhcpserverIP, msg.sndr,
                        DATA({HardwareAddress  |-> msg.data.hwAdr(),
                                 NewAddress |-> newAdr}), dhcpoffer)
    else
      dhcpserver.Output(dhcpserverIP, broadcast,
                        DATA({HardwareAddress  |-> msg.data.hwAdr(),
                                 NewAddress |-> newAdr}), dhcpoffer)

  DHCPServerGetMailbox() as String =
    dhcpserver.mailbox.asString()

  DHCPServerDeleteMsg(id as String) =
     let msg = IdToMSG(id)
     dhcpserver.mailbox(msg) := false
     MESSAGEs(msg) := false
```

7.2.11 Fire

```
class UPnPModelServer...
  Fire() =
    try
      RunUPnP()
    catch
      e as Object : POPUP("Exception in RunUPnP: " + e.asString())

  NETWORK_Fire(netw as String) =
    try
      IdToNetwork(netw).RunNetwork()
    catch
      e as Object : POPUP("Exception in RunNetwork: " + e.asString())

  CONTROLPOINT_Fire(ctrl as String) =
    try
      IdToControlPoint(ctrl).RunControlPoint()
    catch
      e as Object :
        POPUP("Exception in RunControlPoint: " + e.asString())

  DEVICE_Fire(dev as String) =
    try
      IdToDevice(dev).RunDevice()
    catch
      e as Object : POPUP("Exception in RunDevice: " + e.asString())
```

7.2.12 Step

The main rule of the simulation ASM.

```
class UPnPModelServer...
  Step() =
    try
      RunUPnP()
      now := now + 1
    catch
      e as Object : POPUP("Exception in RunUPnP: " + e.asString())
```

7.2.13 Initialization

The set of all possible control point/device addresses is given by the following static functions.

```
deviceAddressSpace as Set[ADDRESS] =
  {ADDRESS(deviceNetId,
           deviceNetId + ".1." + asString(i), 0) | i in {1..100}}
controlPointAddressSpace as Set[ADDRESS] =
  {ADDRESS(controlPointNetId,
           controlPointNetId + ".2." + asString(i), 0) | i in {1..100}}
```

The available addresses are given by the following derived functions.

```
availableDeviceAddresses() as Set[ADDRESS] =
  {a | a in deviceAddressSpace where
                     not (exists d in DEVICEs where d.adr = a)}
availableControlPointAddresses() as Set[ADDRESS] =
  {a | a in controlPointAddressSpace where
                     not (exists c in CONTROLPOINTs where c.adr = a)}
```

7.2.13.1 Device network communicator

```
deviceNetId as String = "1.1"
deviceNetwork as COMMUNICATOR =
  new COMMUNICATOR(deviceNetId,
              //*** addressTable
              //local addresses
              {devices |-> deviceAddressSpace} merge
              {d |-> {d} | d in deviceAddressSpace} merge
              {dhcpserverIP |-> {dhcpserverIP}} merge
              //nonlocal addresses
              {controlPoints |-> {controlPoints}} merge
              {c |-> {c} | c in controlPointAddressSpace},
              //*** routingTable
              undef)
```

7.2.13.2 ControlPoint network communicator

```
controlPointNetId as String = "2.2"
controlPointNetwork as COMMUNICATOR =
```

```
  new COMMUNICATOR(controlPointNetId,
             //*** addressTable
             //local addresses
             {controlPoints |-> controlPointAddressSpace} merge
             {c |-> {c} | c in controlPointAddressSpace} merge
             //nonlocal addresses
             {d |-> {d} | d in deviceAddressSpace} merge
             {devices |-> {devices}},
             //*** routingTable
             undef)
```

7.2.13.3 DHCP server

There is a single DHCP server and it is connected to the device network.

```
dhcpserverIP as ADDRESS = ADDRESS(deviceNetId, deviceNetId+".10.10",0)
dhcpserver as DHCPSERVER = new DHCPSERVER(dhcpserverIP,deviceNetwork)
```

7.2.13.4 The initialization rule.

```
class UPnPModelServer...
  Initialize() =
    now             := 0
    COMMUNICATORs   := {deviceNetwork, controlPointNetwork}
    DHCPSERVERs     := {dhcpserver}
    deviceNetwork.routingTable :=
      {controlPoints |-> controlPointNetwork} merge
      {c |-> controlPointNetwork | c in controlPointAddressSpace}
    controlPointNetwork.routingTable :=
      {devices |-> deviceNetwork } merge
      {d |-> deviceNetwork | d in deviceAddressSpace}
```

7.2.14 Reinitialize

```
class UPnPModelServer...
  Reinitialize() =
    DEVICEs             := {}
    CONTROLPOINTs       := {}
    MESSAGEs            := {}
    deviceNetwork.mailbox := {}
    controlPointNetwork.mailbox := {}
    dhcpserver.mailbox := {}
    now                 := 0
```

7.2.15 other

```
class UPnPModelServer...
  Get_Networks() as String =
    asString(COMMUNICATORs)
  NETWORK_GetType(netw as String) as String =
    asString(IdToNetwork(netw).netid)
  NETWORK_GetIP(netw as String) as String = ""
```

7.3 Global mappings from identifiers to agents.

7.3.1 IdToDevice

Map an id to the corresponding device.

```
IdToDevice(devId as String) as DEVICE =
  unique dev | dev in DEVICEs where asString(dev) = devId
```

7.3.2 IdToNetwork

Map an id to the corresponding network.

```
IdToNetwork(netwId as String) as COMMUNICATOR =
  unique netw | netw in COMMUNICATORs where asString(netw) = netwId
```

7.3.3 IdToControlPoint

Map an id to the corresponding control point.

```
IdToControlPoint(ctrlId as String) as CONTROLPOINT =
  unique ctrl | ctrl in CONTROLPOINTs where asString(ctrl) = ctrlId
```

7.3.4 IdToMSG

```
IdToMSG(msgId as String) as MESSAGE =
  unique m | m in MESSAGEs where asString(m) = msgId
```

7.3.5 IdToSrvc

The id of a service is unique within the scope of a given device.

```
IdToSrvc(dev as DEVICE, sId as String) as ExtSERVICE =
  (unique (srvc as ExtSERVICE) |
         srvc in dev.srvcs where (srvc as ExtSERVICE).GetId() = sId)
```

7.4 External Functions

7.4.1 guessCandidateAdr

External function that returns a new IP address for the given device.

```
guessAutoIPAdr(d as DEVICE) as ADDRESS =
  choose a in availableDeviceAddresses() do
    return(a)
  ifnone
    return(thisDevice)
```

7.4.2 PrintResponse

External function that prints the response from a device in the control point.

```
PrintResponse(ctrl as CONTROLPOINT, msg as MESSAGE) =
  skip
```